Mohammad Thabet

Modeling Time-critical Tasks for Heterogeneous Robotic Systems in Programming by Demonstration

School of Electrical Engineering
Department of Electrical Engineering and Automation

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Technology

Espoo, August 18, 2015

Instructor: D.Sc. Alberto Montebelli
Aalto University
School of Electrical Engineering

Supervisors: Professor Ville Kyrki
Aalto University
School of Electrical Engineering

Professor Thomas Gustafsson
Luleå University of Technology

Acknowledgments

The work that culminated in this thesis would not have been possible if it weren't for the support, both moral and material, of the Intelligent Robotics group at Aalto. Being part of that group meant having access to state-of-the-art equipment, and most importantly, the exceptional expertise of the members which provided ample opportunities to learn. I would especially like to thank my supervisor and head of the group Ville Kyrki for his invaluable support and guidance. I learned and am still learning a lot from him. I also wish to extend my gratitude to my instructor Alberto Montebelli. Apart from his endless support and our technical discussions, I have enjoyed our long talks about philosophy and other matters. In fact, the opportunity to work with Ville and Alberto is one of the main reasons I chose this topic for my thesis.

Being a Spacemaster was a spectacular experience in which I got to travel all over Europe and study in three prestigious universities. My fellow Spacemasters were what made that adventure even more enthralling, especially those who came to Helsinki as well: Rokon, Rian, Kashif, Bagus, Johnny, Rachit, David, and Martin, and I am thankful to have had the company of such outstanding people. The long nights I stayed working at the university were made bearable by the company of Martin and David. I will never forget the good times I had with them, especially our small excursions to the terrace where we often came up with various (potentially game-changing!) ideas. I could not ask for better friends.

I wish to thank the European Education, Audiovisual and Culture Executive Agency (EACEA) for granting me the Erasmus Mundus scholarship that allowed me to participate in the Spacemaster program.

I am also thankful for my family and friends back home. Their support and encouragement were of great importance to me as always.

Finally, I dedicate this thesis to my mother. I owe her all my achievements, and I could never repay her for all that she has done for me over the years.

Espoo, August 18, 2015

Mohammad Thabet

Aalto University
School of Electrical Engineering Abstract of the Master's Thesis

Author:	Mohammad Thabet
Title of the thesis:	Modeling Time-critical Tasks for Heterogeneous Robotic Systems in Programming by Demonstration
Date:	August 18, 2015 **Number of pages:** 64+8
Department:	Electrical Engineering and Automation
Programme:	Master's Degree Programme in Space Science and Technology
Professorship:	Automation Technology (AS-84)
Supervisors:	Professor Ville Kyrki (Aalto)
	Professor Thomas Gustafsson (LTU)
Instructor:	D.Sc. Alberto Monetebelli

Programming by demonstration has been introduced in recent years as a rapid and efficient way to impart skills to robots. In programming by demonstration, a robot learns a new skill by having an end-user perform demonstrations of the skill, bypassing the need for traditional programming. As robotic systems can often be considered as composed of multiple heterogeneous components, learning skills for these systems requires capturing and preserving concurrency and synchronization requirements in addition to task structure. Furthermore, learning time-critical tasks depends on the ability to model temporal elements in demonstrations. This thesis proposes a modeling framework in programming by demonstration based on Petri nets capable of modeling these aspects. In this approach, models of tasks are constructed from segmented demonstrations as task Petri nets, which can be executed as discrete controllers for reproduction. The implementation details of a complete prototypical system are given, showing how elements of time-critical tasks can be mapped to those of Petri nets. Finally, the approach is validated by an experiment in which a robot learns and reproduces a musical keyboard-playing task.

Keywords: programming by demonstration, imitation learning, Petri nets, time-critical, heterogeneous robots

Contents

List of Figures

List of Abbreviations

CAN	Controller Area Network
CFG	context-free grammar
CPN	colored Petri net
DEDS	discrete-event dynamic system
DMP	dynamic movement primitives
DoF	degrees of freedom
DTW	dynamic time warping
EM	expectation maximization
ETE	event time error
FRI	Fast Research Interface
FSM	finite state machine
GMM	Gaussian mixture model
GMR	Gaussian mixture regression
HMM	hidden Markov model
IETE	inter-event time error
KRC	KUKA Robot Controller
KRL	KUKA Robot Language
LCS	longest common substring
LWL	locally weighted learning
NURBS	non-uniform rational B-splines
PbD	programming by demonstration
PHMM	parameterized HMM

PID controller	proportional-integral-derivative controller
PN	Petri net
SCFG	stochastic CFG
SVM	support vector machines
TPN	task Petri net
UDP	User Datagram Protocol

Chapter 1

Introduction

Although robots are increasingly being utilized in various roles, ranging from industrial applications to space exploration, they are still far from being ubiquitous. Presently, robots are largely confined to performing predefined movements in controlled environments, whether in labs or assembly lines. These robots are programmed for just one very specific task, and they often operate in isolation from people. Conversely, general-purpose robots that can work with humans need to be able to perform a multitude of different tasks dexterously and safely in generic environments. Such robots can be manufactured given current hardware technology. However, the state of the art in software still imposes great limitations on robot behavior.

The limited capabilities of robots are generally due to the manner in which they are programmed. Robots are traditionally programmed by experts using hand-written programs that can only perform one very controlled task, a process that has two major drawbacks. First, teaching a robot a new skill becomes very costly and time-consuming as only expert programmers can write a program tailored for this specific skill. Second, such programs usually have very poor adaptability, and even a slight change in the task requirements will often cause them to fail (e.g. due to external disturbances). This effectively precludes robots from being employed in everyday chores, and thus motivates a new approach to robot programming.

Programming by demonstration (PbD), also called imitation learning or learning from demonstration, is a relatively recent approach that promises solutions to these problems. In PbD, robots are taught new skills or tasks simply by having a human teacher (or possibly another robotic agent) perform demonstrations, which they capture using various sensors. The robot then makes use of any of numerous learning algorithms to learn these skills and generalize upon them. After learning, the robot is able to reproduce the skill with a measure of adaptability. PbD thus allows for rapid teaching of new skills to robots, as a skill can typically be learned from just a few demonstrations. Furthermore, it makes the teaching process accessible to virtually everyone, since it eliminates the need for a computer expert to explicitly program the skill, and the required expertise is limited to the task to be learned.

One class of tasks that is of special interest to this thesis is *time-critical tasks*. These are tasks that require accurate timing of their component actions

to be considered successful, analogous to hard real-time processes in computer software. Therefore, a learner must not only learn *what* to do, but also exactly *when* to do it. An example of time-critical tasks would be playing musical instruments, where it is as much important to play the notes in a timely fashion as it is important to play the correct notes. Consequently, learning these task requires more powerful modeling paradigms than is usually employed.

1.1 Objectives

The goal of this thesis is to develop a general modeling framework in PbD that fulfills two requirements. The first requirement is the ability to effectively model time-critical tasks. The vast majority of approaches in PbD to date have been reactive in nature, meaning that action selection depends solely on the current state of the system or the environment with no regard for past events or states [1]. Reactive methods, although they usually yield good performance, are not adequate to model repetitive or time-critical tasks since they discard temporal data.

The second requirement is to learn tasks for heterogeneous robotic systems, or robots that are composed of multiple parts that can move independently. This is an issue that is left largely unexplored in the literature, as robots are generally treated as a single atomic entity in the context of PbD. However, by considering each part separately, it becomes possible to learn complex movements of the entire robot as a collection of simple movements of its components, which can reduce the complexity of the learning process. Therefore, the modeling framework to be developed must also be capable of modeling concurrent movements of several components and synchronization requirements between them.

To this end, a modeling framework based on Petri nets is proposed by the thesis. A complete system was developed to learn Petri net models of tasks from demonstrations and use them to reproduce the tasks. Finally, an experiment is set up where the system learns and reproduces a musical passage on a keyboard to test and verify the capabilities of the system.

1.2 Outline

To give a sense of place for the contribution of the thesis in state-of-the-art research, Chapter 2 discusses approaches to PbD prevalent in the literature. Since the proposed approach is based on Petri nets, Chapter 3 is dedicated entirely to the basics of Petri nets as well as some of their relevant extensions. Chapter 4 presents implementation details of the developed system, and gives an example of how the Petri net approach can be implemented. An experiment is detailed in Chapter 5 to test and evaluate the performance of the system. Finally, Chapter 6 discusses the conclusion of the thesis and future work.

Chapter 2

Programming by Demonstration

Programming by demonstration has recently emerged in the robotics scene as a viable alternative to traditional programming that should allow rapid programming of robots. In this chapter, current literature discussing approaches to task learning, methods of providing demonstrations, and modeling methodologies in PbD will be reviewed.

2.1 Approaches to Task Learning

Approaches to robot PbD can be classified into two categories based on the level of task representation: trajectory-level learning and symbolic-level learning [2]. At trajectory-level learning, the task as a whole is encoded as a trajectory and generalized from multiple demonstrations using statistical or aggregation methods. The encoding can be done in joint space, task space, or torque space, and usually involves dimensionality reduction techniques. On the other hand, learning at the symbolic level requires segmentation of the task into a series of predefined actions. Learning the skill hence comprises establishing relationships between those actions, and encoding them in a concise model. Figure 2.1 highlights the difference between the two approaches.

2.2 Dataset Acquisition Methods

For a robot to be able to imitate a demonstration, a mapping between the teacher actions and those of the learner must be found; a problem known as the correspondence problem [3]. In [1], this mapping is further divided into two: record mapping and embodiment mapping. Record mapping refers to whether the learner records the actions of the teacher as exactly experienced by them, or through some other manifestation of those actions (e.g. by externally observing the teacher). Embodiment mapping refers to the relationship between the dataset of teacher actions recorded by the learner and the action the learner will later reproduce. For example, a robotic learner can record the actions of a human demonstrator but might not be able to reproduce them in exactly the same manner because the physical configuration of the learner is different, and thus the same actions have different embodiments in both agents.

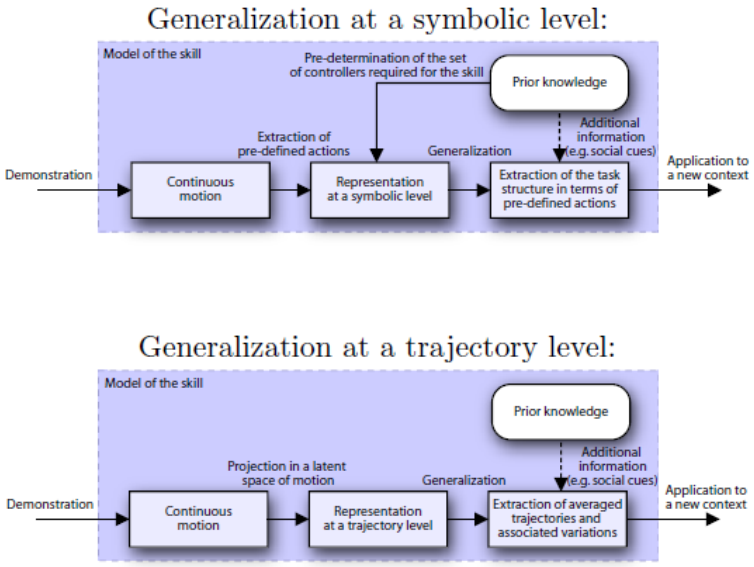


Figure 2.1: Levels of skill representation. Source: [2]

According to Argall et al. [1], dataset acquisition by the learner is categorized into four methods based on the record and embodiment mappings (Figure 2.2). The first method is *teleoperation*, where the teacher moves the robot directly using either some form of remote controller or kinesthetically by grabbing the robot and moving it as desired. In teleoperation, the learner records the movement of its own body using its sensors, and thus both the record and the embodiment mappings are the identity mappings. Second is *sensors on teacher*, in which the robot learner records data from sensors attached to the teacher's body. The third method is *shadowing*, where the robot learner actively tries to mimic or shadow the teacher during the demonstration and records its own movements in the process (for example following a teacher through a sequence of markers). Lastly, in *external observation*, the robot observes the behavior of the teacher through sensors external to the teacher, typically cameras.

Kinesthetic teaching, which is the mode of demonstration used in the thesis work, offers three important advantages over other teaching methods. First, it has direct record and embodiment mappings, which means that the overhead of finding these mappings is eliminated. Second, it offers a natural medium for humans to teach, since this is the primary method humans teach each other new skills. Third, it allows for demonstrating forces and torques along with trajectories [4, 5], which would be rather difficult to do if the demonstrator did not have direct contact with the learner.

Kinesthetic teaching has gained momentum in recent years due to advancements in robot hardware and software control. The advent of light-weight robots fitted with torque sensors at the joints, like the KUKA LWR+ [6], made it possible to actively control the robot by moving it by hand. This, coupled with active gravity compensation on the robot, allows for easy and natural steering of robots during demonstrations.

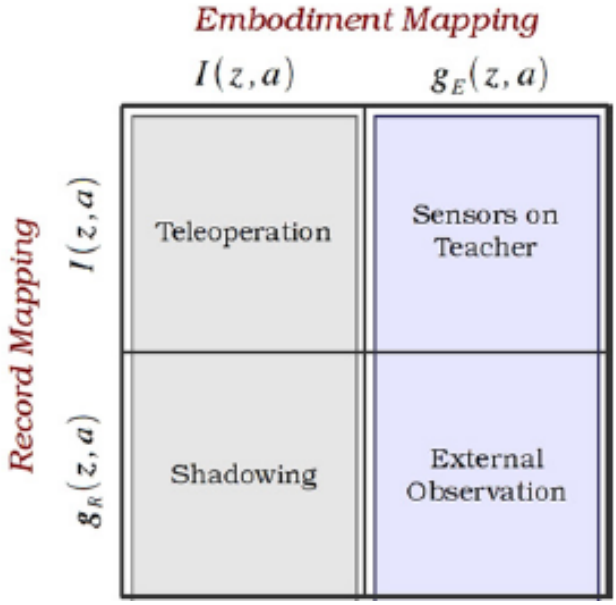


Figure 2.2: Methods of dataset acquisition by the learner. $I(z,a)$ represent an identity mapping function, while $g_E(z,a)$ and $g_R(z,a)$ represent some other embodiment and recording mapping functions, respectively. Figure adapted from [1].

2.3 Trajectory Learning

Trajectory-level learning considers a task as one continuous movement trajectory, or a sequence of sub-tasks each with a goal along the trajectory. It is arguably the most direct way to tackle the problem of modeling tasks, as it only deals with how to perform the task rather than rather than trying to interpret the actions and discern goals. Approaches to encoding skills at the trajectory level can be divided into two classes of methods: statistical modeling and dynamic systems modeling.

2.3.1 Statistical Modeling

In this approach, statistical methods are used to deal with the variability in multiple demonstrations, and to generalize upon them by extracting important, task-defining features from them. Earlier approaches encoded positional trajectories using splines and Bezier curves. For example, [7] used a stereo vision setup to estimate the pose of an object at discrete points along a path. The trajectory was then reconstructed in task space or joint space using a regression technique based on smoothing vector splines. Only one demonstration was used to learn the curve however.

Generalization of a task necessitates multiple demonstrations. In [8], multiple human demonstrations were used to identify a range of acceptable motion and force trajectories to achieve a pick and place task. Inconsistencies across the demonstrations were used to identify the accuracy requirements of the task, effectively forming a boundary region in which the robot can perform the move-

ments. A different approach to the same effect was given in [9], in which multiple demonstrations were used to reproduce a pick and place task in a virtual environment. Trajectories of different demonstrations were clustered using a distance-based algorithm, then for each cluster the most consistent trajectories were selected using hidden Markov models (HMMs) and approximated using non-uniform rational B-splines (NURBS).

A wide range of machine learning techniques has also been used to encode trajectories. Schaal et al. in [10] used locally weighted learning (LWL) techniques to teach robots juggling and pole-balancing tasks. The key concept behind their methods is to approximate nonlinear functions using piecewise linear models, and to determine a region of validity where a local model holds, and how to fit it in this region. Two variants of LWL algorithms were discussed: memory-based LWL, in which the system stores all the training data and then uses lookup and interpolation techniques when a prediction to a new input is to be generated; and incremental LWL, in which each new data point is incrementally incorporated into an initial model and used to update its parameters.

Some especially popular modeling techniques are those based on Gaussian processes, namely Gaussian mixture models (GMMs) and HMMs. GMMs encode trajectories as a linear superposition of multi-dimensional Gaussian components that also have a temporal dimension [11]. The means and covariance matrices of the Gaussians can be learned from demonstrations. For instance, [12] uses a GMM to encode trajectories of a beverage-pouring task. dynamic time warping (DTW) is used to temporally normalize trajectories in different demonstrations, which are then used to train the GMM using an expectation maximization (EM) algorithm. The trajectories are encoded in task space to bypass the correspondence problem. Gaussian mixture regression (GMR) is used in conjunction with an attractor-based trajectory optimization scheme to reproduce task-space trajectories, which are finally projected onto joint space via a kinematic model.

While GMMs model time explicitly (by virtue of the temporal dimension of the Gaussians), HMMs do not. This can be advantageous since the models can be modulated to produce similar trajectories in different areas of the workspace [13]. Like GMMs, an HMM consists of a mixture of Gaussians called states, but is also associated with a transition matrix that represents the probabilities of switching between a states. Unlike GMMs, the choice of the Gaussian component is not solely dependent on the observation, but is also dependent on the choice for the previous observation (i.e. on the previous state) [11]. In [13], Calinon et al. used an HMM model to encode trajectories in task space, and the Baum-Welch algorithm to learn its parameters. For reproduction, velocity commands were estimated through GMR.

2.3.2 Dynamic System Modeling

In this approach, nonlinear dynamical systems are used to model trajectories. This is usually achieved by first obtaining a set of linear differential equations that represent a simple dynamical system (e.g. a spring-damper system), and then transform it into a nonlinear system using a forcing term that can be

learned from demonstrations. The resulting system is associated with predefined attractor dynamics that govern the asymptotic behavior. Point attractors are used for discrete movements (e.g. reaching), while limit cycle attractors are used for rhythmic movements (e.g. drumming) [2].

The variant of this approach that is most relevant to state-of-the-art research is dynamic movement primitives (DMP). DMP is a design principle that centers around three main components. The first of those is the canonical system, which is a simple dynamical system that generates a behavioral phase variable that acts as a substitute for an explicit time variable. The canonical system can have a point or a limit cycle attractor. The second is the non-linear function approximator that generates the forcing term. The phase variable is used to modulate the nonlinear function approximator, and the resultant forcing term is added to a simple dynamical system to form the third component, the transformation system, which generates state vectors for the robot to follow [14] (see Figure 2.3).

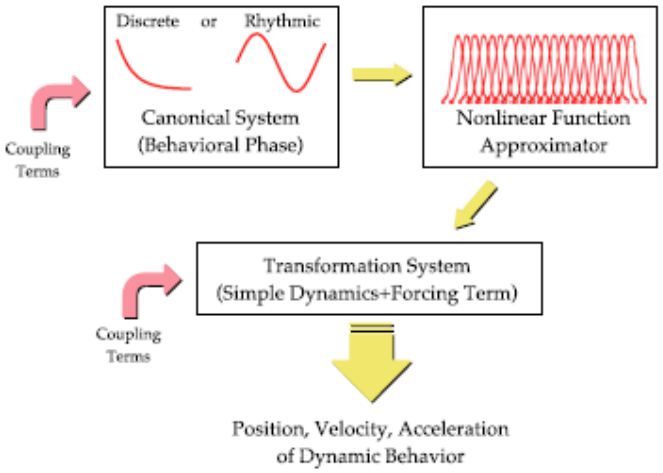


Figure 2.3: Design principle of dynamic movement primitives. Coupling terms are used to modulate the primitives in real-time (e.g. for obstacle avoidance). Source: [14].

DMP have been used successfully to teach robots various tasks, such as wood planing [4], or writing on a notepad [5]. The main advantages of DMP are that it can generate a desired behavior regardless of the initial conditions, that the movement can be appropriated to a new context easily by changing the goal state or the speed of the movement, and that it can also be used to parse observations into sequences of primitives that have been previously learned [14]. This last point is especially useful when considering a hierarchical learning scheme as will be discussed in the following section.

2.4 Hierarchical Symbolic Learning

Learning at the symbolic level involves segmentation of the high-level task into smaller, mid-level actions, and using symbols to represent and abstract them.

These actions can in turn be composed of low-level motor/action primitives, which are simple atomic movements the agent can perform. Learning the task thus becomes a matter of finding structural relationships between the primitives that describe actions, and similar relationships between actions to describe tasks. This essentially creates a hierarchical structure to task learning, both at the symbolic representation level and on the task representation level as a whole.

The rationale of this approach is grounded in how humans reason about actions. There is strong neurobiological evidence that humans perceive actions by other agents as a sequence of motor primitives, and that symbol manipulation in humans is founded upon behavior imitation [15, 16]. The use of symbols to represents action/motor primitives allows more abstract reasoning about the skill in question, which in turn facilitates learning of complete tasks and allows for better generalization. This implies that the robot has to be endowed with a measure of "understanding"[1] of the task at hand, and be able to discern the goals and the intentions of the demonstrator.

The need for segmentation and hierarchical conceptualization of tasks can also be understood from a purely engineering-oriented point of view. Ivanov et al. in [17] argue that simply using statistical pattern recognition techniques is not suitable for activity recognition in certain kinds of domains, namely those that have any of the following properties: insufficient data, where only component examples are available; semantic ambiguity, where semantically equivalent components possess radically different statistical patterns; temporal ambiguity, when different models can segment input at different points in time; and known structure, where the structure of the task is difficult to learn but is known a priori. These conditions lead to considering the problem as divided into two: statistical detection of primitives, and interpretation of the structure that organizes them.

Kruger et al. in [18] identify a three-level action hierarchy: action/motor primitives, actions, and activities. Action/motor primitives are the smallest meaningful movements that can be performed, and form the atomic building blocks that compose actions. Actions are more complex movements that achieve some goal. Lastly, activities refer to the general context of the actions being performed. As an example, if a robot is to be programmed to play tennis, 'forehand', 'backhand', and 'run left' would be primitives; actions would be sequences of these primitives to return a ball or serve; and the activity would be playing tennis in itself.

Kruger et al. also outline different levels of action consideration based on the level of complexity as shown in Figure 2.4 [18]. Simple actions, such as motor primitives, are most suitable to be modeled on the trajectory level. More complex actions, such as composites of primitives, can be modeled on the symbolic level. Finally, a task plan can be extracted as sequences of these composite actions. It should be noted that there exists no consensus in the literature about levels of representation and abstraction of actions and tasks; the appropriate number of levels depends on the complexity of the task and the preference of the system designer. However, the majority of works seem to favor two-level

[1] this is not to imply that the robot is capable of anything similar to human-level understanding.

hierarchies of representation, as they are appropriate for the level of complexity of the activities being studied in contemporary research. For the remainder of this section, the term primitives will be used to denote the low-level component actions, and the terms actions or tasks will be used to denote high-level actions, depending on the scope of the work being discussed.

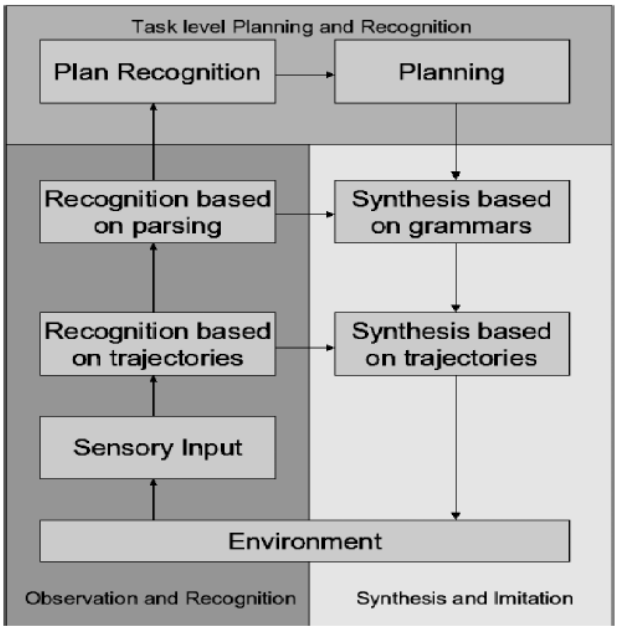


Figure 2.4: Levels of action recognition and synthesis. Source: [18].

2.4.1 Modeling and Recognition of Primitives

The methods of acquiring and representing primitives vary significantly in the literature. The first thing to consider is how the component primitives will be acquired by the system, and whether some prior knowledge will be necessary, usually in the form of a predefined vocabulary or repertoire of primitives. The standard approach is to use primitives that are predefined by the user [19, 20, 21, 22, 23] . Depending on the nature of the action, these primitives can be manually modeled and then recognized in a variety of ways, for example by considering them as states and just detecting their pre- and post-conditions [20] for scene or goal-oriented approaches, or by using any of the trajectory-level modeling techniques discussed in Section 2.3 for movement-oriented approaches. In case simple motor primitives are used as building blocks of actions, one recently popular method for recognition is to classify the movement using support vector machines (SVM) that have been trained using labeled data [23, 24].

Conversely, some research has also been done on unsupervised learning of action primitives from demonstrations that requires no previous knowledge. In [25], Kruger et al. propose a method to automatically define and extract primitives based on their effects on objects in an object manipulation task. First, object trajectories from demonstrations are segmented and modeled as HMMs and then combined into one model. Then, the trajectories are regenerated as

strings of discrete state sequences from the model, and primitives are extracted in the object state space using a longest common substring (LCS) approach (see Figure 2.5). Finally, different hand trajectories that correspond to the primitives in object space (i.e. trajectories that cause the same effect on objects) are grouped together and remodeled as parameterized HMMs (PHMMs) to obtain models of primitives in movement space.

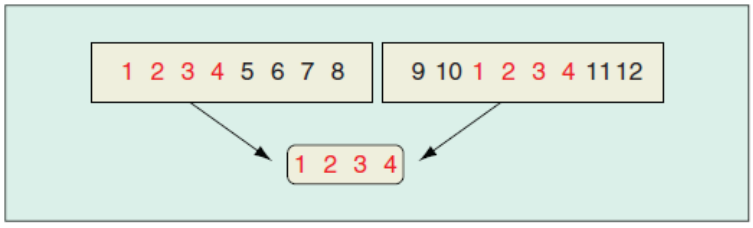


Figure 2.5: Two trajectories represented as sequences of states. The LCS is found to be (1 2 3 4). The set of final primitives extracted are (1 2 3 4), (5 6 7 8), (9 10), (11 12). Source: [25].

This approach, although significantly more complex, is promising for learning tasks in applications in which the domain is undefined, or where it is generally not possible to provide prior information to the system. It is undoubtedly one step closer towards fully autonomous robots. However, if the domain is well-defined, using predefined primitives generally yields better performance. In addition to being simple, it has the advantage of allowing the user to determine the granularity of representation suitable for the application, as well as to define the vocabulary of primitives necessary for efficient modeling.

2.4.2 Modeling on the Symbolic Level

On the high-level end, the vast majority of works on modeling on the symbolic level have used simple directed graphs [21, 22, 23, 26], while some others have used HMMs [19, 24]. However, some recent research is devoted to other approaches, such as syntactic methods using formal grammars [27, 28], or learning Petri net models of tasks [29].

Graph-based Methods

In graph-based methods, tasks are modeled as directed graphs whose nodes represent component actions or primitives. In most cases, these graphs can also be thought of as finite state machines (FSMs), in which the primitives are states. Learning and generalizing tasks consists of merging different graphs corresponding to different demonstrations.

In [20, 26], Nicolescu et al. introduce a graph-based approach to model an object transportation task and teach it to a wheeled mobile robot. Each node in the graph represents an abstract behavior, and the graph is built incrementally during the demonstration by adding to the graph behaviors whose preconditions are met and have been confirmed relative by cues from the teacher. Generaliza-

tion is made on the level of the graph topology by combining graphs of different demonstrations (see Figure 2.6).

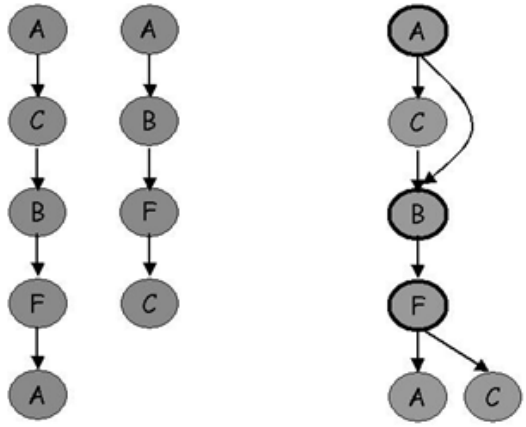


Figure 2.6: An example of generalization on the symbolic level from multiple demonstrations. The two graphs on the left are from two different demonstrations. The graph on the right is the generalized topology. Figure adapted from [26].

Similarly, in [21], an incremental, hierarchical, graph-based approach is used to model manipulation tasks. There, the task is segmented into smaller subtasks, each of which is an abstraction of a sequence of even smaller atomic actions (or elementary operations). Generalization is done by incrementally reducing temporal dependence of subtasks on each other as more demonstrations become available, eventually only retaining the smallest set of dependencies that are consistent among all demonstrations. A very similar approach is employed in [22], where the high-level task is segmented and quantized into states. Generalization again consists of finding the minimum set of constraints on state precedence and dependence that is consistent with all demonstrations. These constraints are then used to generate possible execution sequences for the reproduction of the task.

In a recent work in the same vein, [23] uses kinesthetic demonstrations to teach a robotic hand and arm to unscrew a light bulb. The skill is considered as composed of primitives that have been learned beforehand, and each is modeled as a dynamic system with an attractor behavior. Learning the skill consists of constructing a graph (or state machine) where the nodes (or states) are primitives, and transitions signify switching between primitives, of which only one can be active at a time (see Figure 2.7). Each node is associated with a SVM classifier which has been trained using labeled demonstration data, and decides whether to continue executing the current primitive or switch to the next one based on features obtained from raw sensor data.

Graph-based methods, although simple and intuitive, do not provide a robust mechanism for discarding erroneous parts in demonstrations. They are also reactive and do not handle time constraints in tasks. Furthermore, since they are based on direct acyclic graphs, they are incapable of adequately handling repetition, at least not without ad hoc extensions.

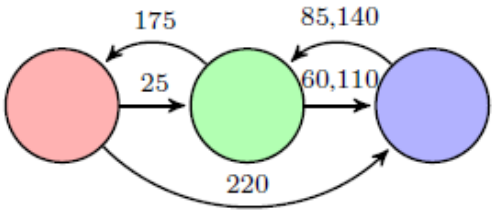


Figure 2.7: Representation of a task as a directed graph taking the form of an FSM. Here each node corresponds to a primitive, and the node color represents which primitive. Transition labels represent the time of transition points in the demonstration. A classifier for each node is then trained using features at transition points to decide when to switch to the next primitive. Source: [23]

Syntactic Methods

Some research that has been recently done on action recognition and modeling focused on stochastic parsing of observations using learned action/activity grammars. The idea is that by learning task structure in the form of an activity grammar, it is possible to use this knowledge to help recognize more complex tasks that share that structure, and to enforce consistency when capturing demonstrations. This can be seen as providing context for later observations, which makes it much easier to disambiguate or discard out-of-context actions whether because of errors in classification, or errors in demonstrations themselves.

In this approach, context-free grammars (CFGs), a type of formal grammars, were used to model higher-level actions as sequences of primitives. CFGs have their origins in linguistics, where they are used to describe the structure of sentences in natural language. A CFG consists of a set of terminals, which in a PbD framework represent primitives; a set on non-terminals, which are abstractions of sequences of terminals; and a set of production rules, which expand non-terminals to terminals and other non-terminals. Note that the term "context-free" is used to distinguish it from a context-sensitive grammar, which is another type of grammar, and does not imply that it is unable to represent contextual knowledge.

In state-of-the-art research, a probabilistic extension of CFGs, stochastic CFGs (SCFGs), are used to model complex actions and tasks. In SCFGs, each rule is assigned a probability parameter, reflecting the probability that a given non-terminal will be expanded into the specified string. In a seminal work, Ivanov et al. [17] used SCFGs to model structured gestures. HMMs were used to detect atomic gestures or primitives, which were then fed as symbols into a stochastic context-free parser to classify the action based on a set of predefined rules and rule probabilities (Figure 2.8). Furthermore, the outputs of these low-level detectors are probabilistic, and the probability of each symbol is considered by the parser. The primary contribution of this work is incorporating input symbol uncertainty into the framework, a practice that has become commonplace in subsequent work.

The same approach has been applied in a PbD framework in [27] to model

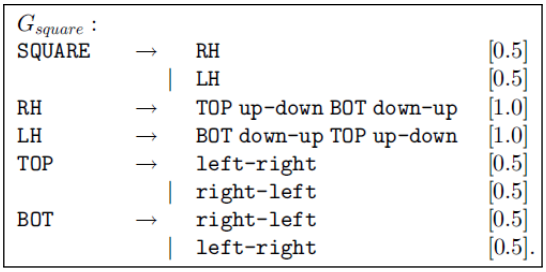


Figure 2.8: A grammar describing a square gesture. The set of production rules are able to describe both a right-handed (clockwise) and a left-handed (anticlockwise) square. Terminals are lower-case and non-terminals are upper-case. Rule probabilities are given in square brackets. Source: [17]

mid-level manipulation actions. HMMs were also used to detect primitives, and an SCFG parser was used to find the rule that best describes the action. After demonstrations, the robot was able to recognize the actions and symbolically reenact them. This work however was only concerned with recognizing actions using SCFGs, as the rules and rule probabilities used by the parser were given in advance. It did not attempt to infer the rules or their probabilities from demonstrations.

A recent work that addresses the issue of learning grammars from demonstration is given in [28]. The process of inducing grammars starts by first building an initial naïve grammar from the demonstration, in which all the detected symbols are appended to the start symbol. Afterwards, a series of substitute and merge operations are applied on the symbols, resulting eventually in the generalized grammar. Figure 2.9 illustrates these steps with an example. This method of grammar induction also incorporates uncertainty in the input symbols, and the rule probabilities are modulated by the symbols probabilities. The approach is validated by an experiment in which a robot was able to infer the rules of a puzzle from training demonstrations of solutions, and later parse noisy test demonstrations of a slightly more complicated variant of the puzzle. The robot was eventually able to disambiguate the test demonstrations and extract the correct sequence of actions of the solution and reproduce it.

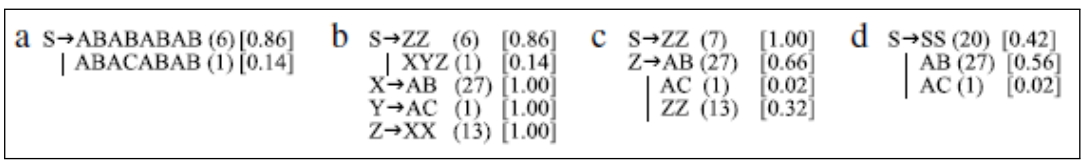


Figure 2.9: An example of how the task structure from 7 demonstrations is discovered and generalized using CFGs. (a) Initial naïve grammar. (b) AB substituted with X, AC with Y, XX with Z. (c) (X,Y) are merged into X, (X,Z) into Z. (d)(S,Z) are merged into S. Rule probabilities are given in square brackets, and number of occurrences in parentheses. Notice the inclusion of an erroneous demonstration in the initial grammar where C replaces B in one occurrence. Source: [28]

In their essence, SCFGs are not much different from HMMs which are them-

selves a kind of probabilistic extension of deterministic graphs and state machines. However, SCFG modeling has many advantages compared to other methods. First, similar to HMMs, the incorporation of input symbol and rule probabilities yields robustness to noise in input as well as erroneous demonstrations. Second, its structure inherently supports hierarchy and repetition. Third, the model is easily understandable by humans, which allows users to manually specify grammars, or bias rule probabilities to reflect prior knowledge about the domain. However, it is not meant as a generative model, and is not suitable for generating an execution sequence. So far it has only been used to parse a demonstration sequence to reject noise and inconsistencies, and it is this parsed sequence that is later reproduced. It could be argued that an execution sequence can be obtained by recursively applying the rules. This however would lead to widely different sequences each time one is generated, since there is no way of deterministically determining how often should recursions be allowed (i.e. how often to expand non-terminals into other non-terminals instead of terminals).

Petri Nets

Petri nets (PNs) are a modeling tool used mainly for discrete-event dynamic systems (DEDSs). They offer a significant improvement on basic FSMs as they can model concurrency, synchronization, and conflict among other typical properties of DEDSs.[2] Although PNs are widely used in robotics generally, they are rarely used to model tasks in PbD.

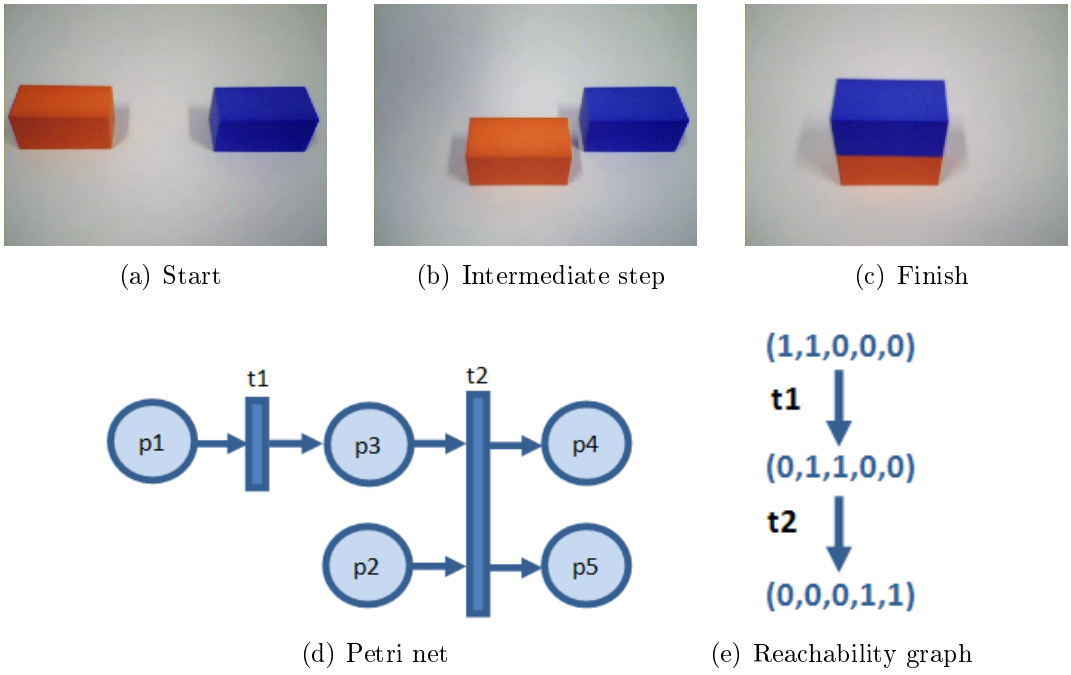
In [29], PNs models of object placement tasks were learned from demonstration. Places in a net represent object states, while transitions represent motions. Object and object state recognition are performed using self-organizing maps, while movement recognition is performed using DMP and affinity propagation. At the beginning of a demonstration, the state of each object is represented by a place in the PN. When a motion finishes, it is added as a transition in the net if it is new and has not previously been performed. Object states before and after a motion are added as input and output places to the transition respectively. For imitation, images of the initial and goal states are obtained and transformed into markings of the PN. A reachability graph is then generated and traversed to find the shortest sequence of transition firings from the initial marking to the goal marking. Finally, this sequence is executed as movements to perform the task.

Figure 2.10 shows this approach applied to a task consisting of stacking two objects. After the PN and the reachability graph are constructed from a demonstration, the movements corresponding to the sequence of transitions found in Figure 2.10(e) can be executed to reproduce the demonstration.

Since this approach is scene-based, it is only capable of modeling actions that produce a noticeable change in the environment. This severely limits the number of applications in which it can be used. It also does not address the issues of modeling time or performing multiple actions simultaneously and synchronizing them. In fact, at least in such simple applications, this approach does not provide any significant advantages over other graph-based approaches.

[2]For a review of Petri nets, see Chapter 3.

The markings of the PN model can be easily mapped to states in an FSM, and transitions would still function similarly.



(a) Start (b) Intermediate step (c) Finish

(d) Petri net (e) Reachability graph

Figure 2.10: Modelling an object placement task by Petri nets. (a) through (c) depict video frames of object states. In (d) place $p1$ corresponds to the red block on the left, $p2$ to the blue block on the right, $p3$ to the red block in the center, $p4$ to the red block in the center with a block above it, and $p5$ to the blue block in the center with a block under it.

2.5 Summary and Discussion

Programming by demonstration is a promising approach to impart skills to robots. It has the potential to ease the many limitations associated with traditional robot programming. This chapter reviewed methods of dataset acquisition in PbD, as well as levels of task representation, with emphasis on symbolic level learning. It has been shown that while simple movements can be learned at the trajectory level, complex tasks require modeling at the symbolic level. The most important and relevant trajectory level techniques were briefly reviewed (for a comparison of the performance of various techniques see [13]). DMP are especially interesting since they can easily be adapted to various contexts and can also be used for detecting primitives. Approaches to modeling and learning on the symbolic level were also covered in some detail, with focus on the most well-established method: graph-based modeling.

Graph-based methods are a natural and intuitive way to model tasks. However, problems can arise when trying to model repetitions and loops since they are based on directed acyclic graphs. Furthermore, they cannot handle parallel execution of primitives and strict timing requirements. In fact, the issue of han-

dling time is left largely unexplored in literature dealing with learning on the symbolic level. All the approaches proposed so far simply sequence primitives one after another, and switching is made either when the currently executing primitive finishes, or according to changes in some feature in the environment or robot state.

While it can be argued that these deficiencies in graphic models can be remedied by ad hoc extensions, trying to extend a simple model to address all these problems at the same time might lead to an unnecessarily contrived model that is inefficient to work with. It is therefore desired to look for a well-established and understood modeling technique that can handle these issues. Petri nets certainly seem a possible answer to that, since they can model concurrency and repetition, have a complete mathematical formulation, and can be extended in a variety of ways to handle multiple aspects such as timing and synchronization [30]. Although they are essentially an extension of basic state machines, they have not been considered in a PbD framework except in very few works (e.g. [29]), none of which seem to capitalize on the full potential of the modeling power of Petri nets.

Chapter 3

Petri Nets

Modeling time-critical tasks at the symbolic level requires not only capturing the task structure, but also the temporal relationship between task components. Furthermore, for heterogeneous robotic systems whose components can move independently, it is imperative to synchronize these components to perform the task correctly and in a timely manner. It is therefore necessary to go beyond the practice of sequencing primitives as simple state machines prevalent in the literature, and to employ a modeling paradigm capable of handling such requirements.

In fact, if task models are to be conceived of as symbolic-level task structures of underlying trajectory-level movement primitives, they can be be approached as hybrid dynamic systems: systems that exhibit continuous as well as discrete dynamics. In a task model, the continuous dynamics are encoded as movement primitives, while the discrete dynamics are encoded in the task structure, since it controls the switching between primitives. The problem of modeling the task structure thus becomes a matter of finding the appropriate discrete-event system modeling framework capable of handling the aforementioned requirements. This thesis proposes Petri nets as the basis of such framework.

Petri nets (PNs) are a mathematical modeling tool (language) mainly used to model DEDSs. They combine a rigorous mathematical formulation with an intuitive graphical representation, which ensures accurate modeling and analysis, as well as easy visualization and interpretation of models [31, 32]. Furthermore, they are vastly extensible beyond their basic form, and have been well-studied and analyzed for decades. This affords PNs with superior modeling power that can even model other types of systems (e.g. continuous and hybrid systems) [30].

PNs have been widely used in robotics, for example in single-robot task specification to coordinate task primitives [33, 34], coordination of dual-arm manipulation tasks [35], and in multi-robot plans [36]. However, PNs have rarely been applied in a PbD setting, as discussed in Section 2.4.2.

In this chapter, the basics of Petri nets will be briefly reviewed, followed by some of their extensions. This serves as preliminary material to the introduction of the concept of task PNs proposed by this thesis and discussed in Chapter 4. Therefore, only information pertaining to task PNs as applied to PbD will be discussed.

3.1 Basics of Petri Nets

A PN consists of four types of elements: places, transitions, arcs, and tokens (Figure 3.2). *Places* are represented by circles, while *transitions* are represented by bars (or a box in some texts), and together they constitute two types of nodes in a PN. *Arcs* are directed and connect places to transitions or vice versa, but never two nodes of the same type. A PN is thus a bipartite graph, in which places and transitions alternate on a path made of arcs. A place is an input place to a transition when there is an arc from that place to the transition, and is an output place to it when there is an arc from the transition to the place. Similarly, a transition is an input transition to a place when there is an arc from the transition to the place, and is an output transition to it when there is an arc from the place to the transition.

(a)

(b)

Figure 3.1: A simple Petri net that has two places, P0 and P1, and one transition, T0. P0 has a single token, while P1 has none. The numbers on the arcs are their weights. (a) before T0 fires, and (b) after T0 fires.

A place can have zero or more *tokens* inside it, which are represented by solid dots. In basic PNs, tokens are indistinguishable from one another. A transition is *enabled* when each of its input places has at least a certain numbers of tokens. This number is given by the *weight* of the arc from that place to the transition, and is represented graphically by a number inscription on the arc. An enabled transition can *fire*, consuming a number of tokens from each of its input places given by the corresponding arc wight, and depositing a number of tokens in each of its output places given by the wight of the corresponding arc. The number of consumed token and that of the deposited tokens need not match.

In Figure 3.1(a), transition T0 has one input place, P0, and one output place, P1. T0 is enabled because P0 has one token, which is equal to or greater than the weight of the arc from P0 to T0, which is also one. When T0 fires, it will consume the token in P0 and deposit two tokens in P1, since the weight of the arc from T0 to P1 is two. Figure 3.1(b) shows the net after T0 fires.

The significance of a place in a net can be interpreted in a number of ways, and consequently the tokens it holds as well. If a place is interpreted as a type of resource in the system being modeled, then the number of tokens it has can denote the amount of that resource available. If it is interpreted as a condition,

then the presence of a token in it represents whether this condition is met or not.

The state of the overall system is thus given by the distribution of tokens in all places. This distribution is known as the *marking* of a PN. The dynamics of a system modeled by a PN is therefore encoded in the evolution of the marking. The marking changes when any transition fires, since this changes the distribution of tokens in the net.

3.1.1 Modeling Power

The ability of PNs to effortlessly model typical characteristics exhibited of DEDSs is what makes them such an effective modeling tool. These characteristics include:

Sequential execution: In Figure 3.2(a), transition T1 can only fire after T0 has fired, since otherwise there will be no token in P1 and T1 will not be enabled. This effectively models the dependence of the process associated with T1 on that associated with T0.

Conflict: In Figure 3.2(b), either T0 or T1 can fire and not both, since if one fires it will consume the token in their common input place, effectively disabling the other. A conflict resolution scheme is required to resolve such a conflict deterministically, for example by assigning priorities to transitions.

Concurrency: Once T0 has fired in Figure 3.2(c), both T1 and T2 can fire at will independent from one another. Thus, both places P1 and P2 can be active at the same time, possible representing two concurrent processes.

Synchronization: In Figure 3.2(d), T0 can only fire if both P0 and P1 have at least 1 token each. Thus, T0 can be interpreted as a synchronization event in which the system has to wait for possibly many concurrent processes to reach a certain point (here given by tokens in P0 and P1) before it advances further.

Limited resource: Real physical systems often have resource constraints, forcing a sub-process to execute only a limited number of times. This situation is modeled as in Figure 3.2(e). Here, if P1 is associated with some resource, then the number of tokens inside it represents the amount of that resource available. Assuming P0 periodically gets a token, T1 can only fire three times, since P1 has just three tokens. Afterwards it becomes disabled and cannot fire. There are of course situations when a resource can be replenished. To reflect this in the example, P1 would have to be the output place of some transition.

3.1.2 Formal Definition

A Petri net is a five-tuple $PN = (P, T, W^+, W^-, M_0)$ such that:

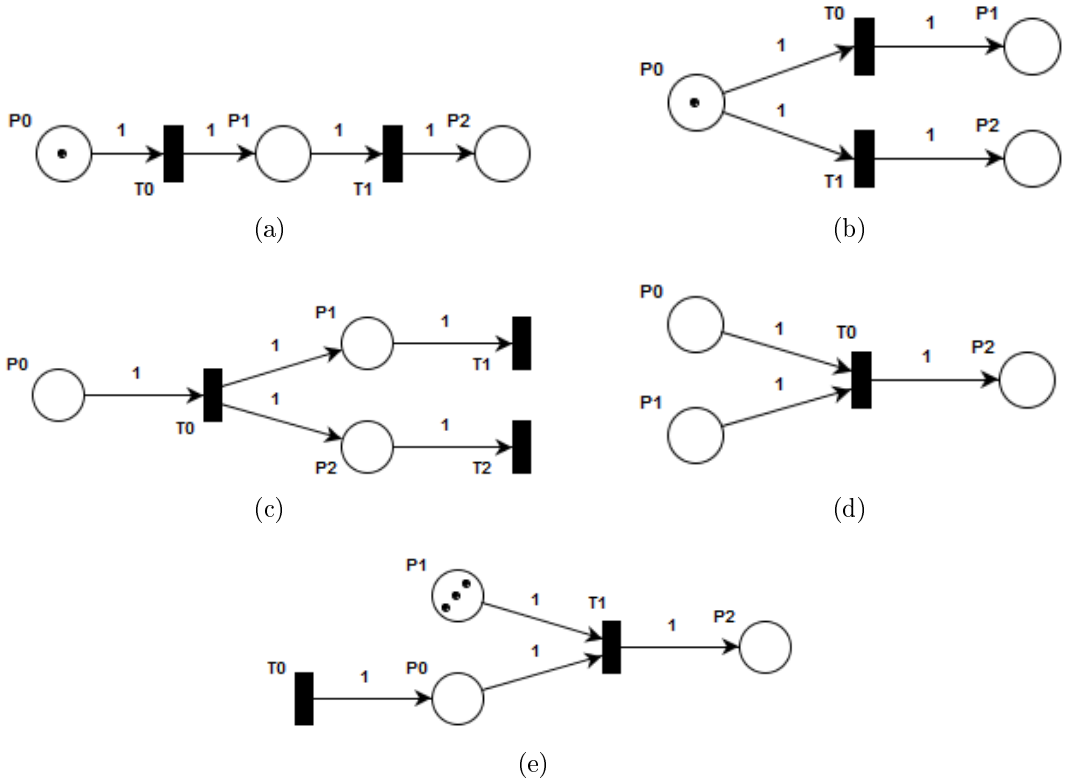


Figure 3.2: Petri net models of some system characteristics. (a) Sequential execution, (b) conflict, (c) concurrency, (d) synchronization, and (e) limited resource.

$P = \{P_0, P_1, \ldots, P_m\}$ is a finite set of places;

$T = \{T_0, T_1, \ldots, T_n\}$ is a finite set of transitions;

$P \cap T = \emptyset$ i.e. the sets P and T are disjoint;

$W^- : P \times T \to \mathbb{N}$ is the input incidence matrix of size $m \times n$;

$W^+ : P \times T \to \mathbb{N}$ is the output incidence matrix of size $m \times n$;

$M_0 : P \to \mathbb{N}$ is the *initial marking* vector of length m.

A Non-zero element of W^- represents an arc from a place to a transition given by the indices of the element, while a non-zero element of W^+ represents an arc from a transition to a place. The value of the element in both cases represents the weight of the corresponding arc. For example, Figure 3.3 shows a PN in which:

$P = \{P_0, P_1, P_2, P_3, P_4\}$;

$T = \{T_0, T_1, T_2, T_3\}$;

$$W^- = \begin{array}{c} \\ P_0 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cccc} T_0 & T_1 & T_2 & T_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}, \quad \text{and} \quad W^+ = \begin{array}{c} \\ P_0 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cccc} T_0 & T_1 & T_2 & T_3 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array};$$

$M_0 = [1, 0, 0, 0, 0]^T$.

The *incidence matrix*, which is important for analysis and simulation as will be seen later, is defined as:

$$W = W^+ - W^-. \tag{3.1}$$

For the PN of Figure 3.3, the incidence matrix is:

$$W = \begin{array}{c} \\ P_0 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cccc} T_0 & T_1 & T_2 & T_3 \end{array} \\ \begin{bmatrix} -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 2 & 0 & -1 & -1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \end{array}$$

3.1.3 Fundamental Equation

For a PN of m places and n transitions, and for some firing sequence which can be performed from marking M_i, the *characteristic vector* s_i is a vector of length n whose j-th component correspond to the number of firings of transition T_j in that sequence. If this firing sequence results in marking M_{i+1}, then the *fundamental equation* (also called *state equation*) is given by:

$$M_{i+1} = M_i + W \cdot s_i. \tag{3.2}$$

Given some marking and a firing sequence achievable from that marking, the fundamental equation is used to calculate the resulting marking. Returning to

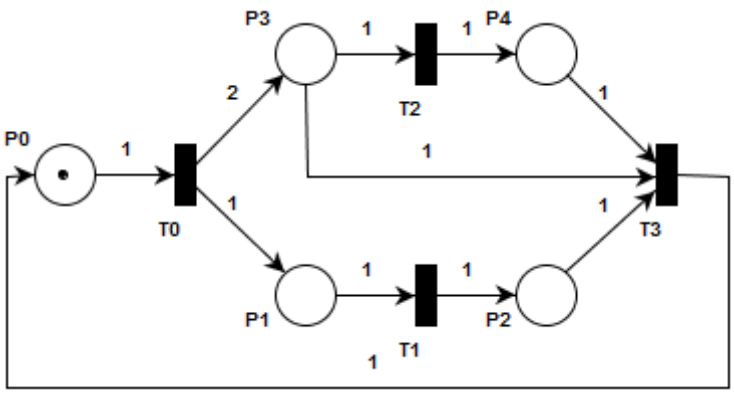


Figure 3.3: A Petri net example.

the example PN of Figure 3.3, if transition T_0 is to fire, then the characteristic vector $s_0 = (1, 0, 0, 0)$, and the new marking is obtained by:

$$\underbrace{\begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix}}_{M_0} + \underbrace{\begin{bmatrix}-1 & 0 & 0 & 1\\ 1 & -1 & 0 & 0\\ 0 & 1 & 0 & -1\\ 2 & 0 & -1 & -1\\ 0 & 0 & 1 & -1\end{bmatrix}}_{W} \cdot \underbrace{\begin{bmatrix}1\\0\\0\\0\end{bmatrix}}_{s_0} = \underbrace{\begin{bmatrix}0\\1\\0\\2\\0\end{bmatrix}}_{M_1}$$

3.2 Extensions

PNs have many extensions that add functional rules to the model in order to increase their modeling power[1], and allow a greater number of applications to be treated. In this section, the extensions relevant to task PNs will be reviewed.

3.2.1 Non-autonomous Petri Nets

In the PNs discussed so far in this chapter, it was shown that a transition *may* fire if it is enabled. These PNs are useful to describe *what* happens when a transition fires, and no assumptions were made as to *when* it will fire. This kind of PNs is known as *autonomous Petri nets*, to reflect the fact that the dynamics of the system are assumed to evolve autonomously. Conversely, *non-autonomous Petri nets* are nets that describe systems whose evolution is conditioned by external events or time. Such PNs can be synchronized and/or timed.

Synchronized Petri Nets

Physical systems often interact with the environment, and their dynamics are influenced by external events emanating from their surroundings. *Synchronized Petri nets* are PNs in which transitions are linked with external events. A transition fires if it is enabled and when a certain external event occurs.

A synchronized Petri net is the triple $(PN, E, Sync)$ such that:

> PN is a Petri net (P, T, W^+, W^-, M_0);
>
> E is a set of external events;
>
> $Sync : T \to E \cup \{e\}$, where $\{e\}$ is the 'always occuring event', is a function that maps transitions to events. $Sync(T_j) = E_j \subset \{E \cup \{e\}\}$ is the set of events on which transition T_j is synchronized.

Synchronized PNs are useful as discrete controllers. These controllers receive information and feedback from external sources, whether from the controlled system, human operators or other controllers. This information can take the

[1] Modeling tool A is said to have greater modeling power than model tool B if tool A can model a greater number of systems than tool B.

form of boolean variables, impulses, or events of other forms. The controller then sends control signals to the environment. These signals can take the form of boolean variables based on the marking of the systems, such that each variable corresponds to whether a place is active (has a token); impulses based on a change in marking; or numerical variables calculated by a data-processing part inside the controller based on the marking. Such systems are also known as *control interpreted Petri nets*.

For example, consider a chemical plant in which the manufacturing process is to be controlled with a control interpreted PN. At a certain stage in the process, a mixer in a tank is mixing its contents, while a heater in a different tank is heating another ingredient. When this ingredient reaches a certain temperature, a draining valve in each tank would open to simultaneously empty their contents into another tank for further processing. Figure 3.4 shows the part of the synchronized PN responsible for this stage of the process. Places are associated with system components, such that a boolean variable is sent to respective actuators based on the the marking of the places. Transition T2 is synchronized on the event that the temperature in the heater tank exceeds a certain value. The event can originate from a data processing part inside the controller that reads continuous sensor data and compares it to threshold values, or it can originate from some other external element. Once the event is triggered, the transition fires, consuming tokens in its input places thus switching off the heater and the mixer, and depositing tokens in its output places thus opening the valves.

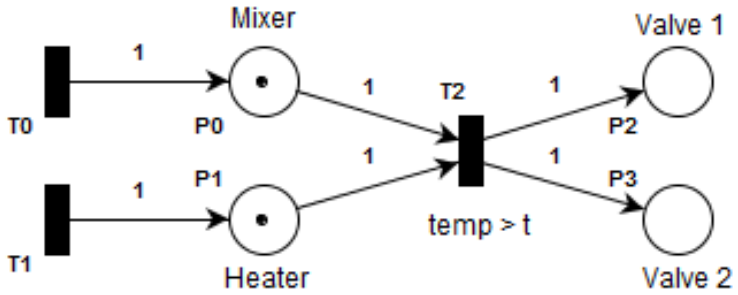


Figure 3.4: A part of a control interpreted Petri net.

Timed Petri Nets

Processes in real physical systems are never instantaneous, and time needs to be introduced in PNs models of those systems to describe their behavior in time. In *timed Petri nets*, a time delay can be associated with either transitions or places. If transitions are associated with time delays, the PN becomes a *T-timed Petri net*, where a transition fires after a certain amount of time has elapsed since the time it was enabled. On the other hand, if places are associated with time delays, the PN becomes a *P-timed Petri net*, where tokens become unavailable for a certain amount of time after being introduced in a place. In either case, time delays are assumed to be determinisitic. There are models however for stochastic time delays, as in *stochastic Petri nets* [30].

T-timed and P-timed PNs are semantically equivalent to one another, and transformation between them is possible [30]. However, T-timed models are more concise and easier to work with, since P-timed PNs require two markings to describe them: one for available tokens, and one for unavailable ones. The focus in this section will therefore be on deterministic T-timed PNs.

A T-timed Petri net is the pair $(PN, Tempo)$ such that:

> PN is a Petri net (P, T, W^+, W^-, M_0);
>
> $Tempo : T \to \mathbb{R}^+$ is a function that maps transitions to positive real time delay values. $Tempo(T_j) = \tau_j$ is the time delay associated with transition T_j

Timed PNs are mainly used in performance evaluation of systems, for example by simulating the system and measuring the total time taken to move from one state to another. However, they can also be incorporated in PN discrete controllers if the controlled system needs to perform certain actions at certain points in time, not just as a reaction to some event. In fact, timed PNs are a special case of synchronized PNs, in which transitions are synchronized to events emanating from a system clock.

3.2.2 Colored Petri Nets

In basic PNs, tokens are indistinguishable from one another. This can lead to very large models for certain types of systems. For example, consider a manufacturing system that processes two types of raw material on the same production line. If this system is to be modeled using standard PNs in which places represent processing stages, it would require two nets, one for each type of raw material. The two nets will share the exact same structure, but their marking at any given point will differ. However, if a type can be attached to tokens to distinguish them, only one PN would suffice.

Moreover, it is sometimes useful to propagate information through the system, and to keep some kind of memory of past events. For instance, if material of type A is input to the system in the previous example, it would be transformed into an intermediate component of type K, and finally into a product of type X. However, if material of type B is used, then it would become L and then Y. This kind of information can only be encoded in the same PN if tokens had a type, and if that type could change according to certain rules as the tokens move around the net.

To tackle these issues, *colored Petri nets* (CPNs) [37] were introduced. In CPNs, each token has a data value attached to it called its *color*. Each place is associated with a *color set*, which is a set of all colors a token inside it can assume. As an analogy, if the token is considered as a variable in programming languages, then its color would be the value of the variable, while the color set it belongs to is the data type.

Arcs in a CPN are associated with *expressions*, rather than weights, that govern the behavior of transition firings. For each transition, the expressions on the arcs coming from each input place decide how many token from which color

set it can consume. A transition becomes enabled only if all the expressions on all its input arcs are satisfied. When a transition fires, the color and amount of tokens deposited in each of its output places is determined by the expressions on the arcs to them. Output arc expressions can be arbitrarily complex; they can be a function of consumed tokens and can produce tokens of another color set.

Figure 3.5 illustrates the dynamics of a simple CPN. For transition T0 to be enabled, the expressions on its input arcs must be able to *bind* to colors from the tokens inside their places. In Figure 3.1(a), the variables x and y are defined over the color set $\{b, r, g\}$ and can be replaced by any of these colors. The expression 1^*x evaluates to a single token of any color in the color set that x is defined over, provided that a token of that color is inside P0. Similarly, the expression 2^*y evaluates to a pair of tokens from P1, except that they have to be the same color, since y can assume only one value at a time. Therefore, x can be bound to (i.e. replaced by) b, g, or r, while y can only be bound to b or r.

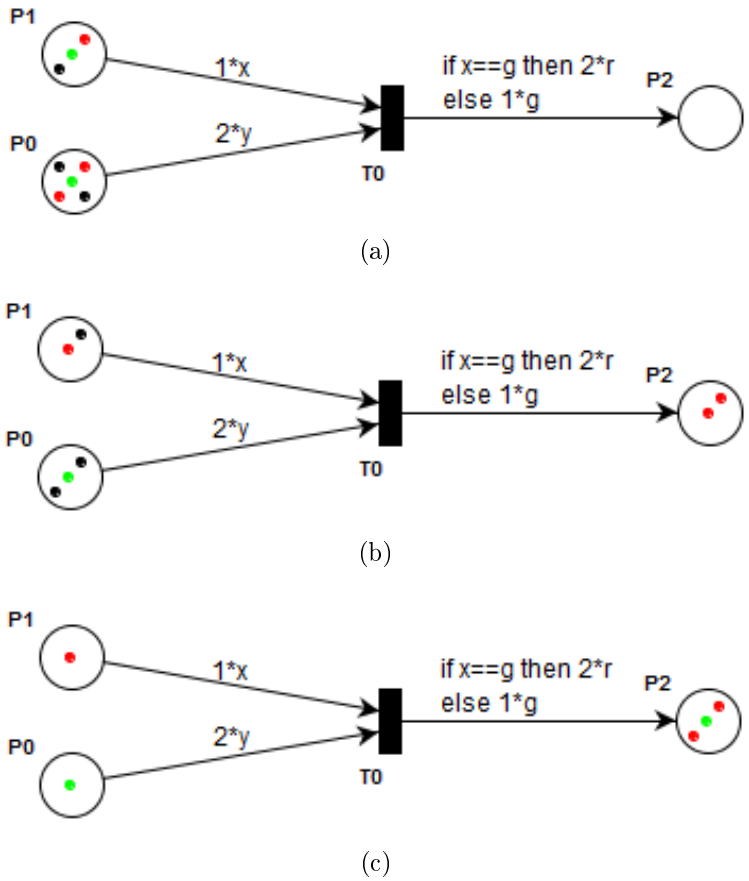


Figure 3.5: A simple colored Petri net. All the place are associated with the color set $\{b, r, g\}$ representing black, red and green tokens respectively. x and y are variables over this color set. (a) Before T0 fires, (b) after T0 fires, and (c) after T0 fires again.

Assuming x is bound to g, and y is bound to r, Figure 3.5(b) shows the

marking after T0 fires. Since x is bound to g, the expression on the output arc evaluates to a pair of red tokens. If both x and y now bind to b, T0 can be fired again as in Figure 3.5(c). This time a single green token is produced as $x \neq g$.

In the previous example, actual colors were used as token colors for illustrative purposes. However, arbitrary color sets can be defined, for example over the set of real values, or even complex data structures. Consequently, arc expressions can be mathematical formulas or complex transformation rules and mapping functions operating on these sets. It is also worth noting that uncolored PNs can be considered a special case of their colored counterparts, where there is only one color set, the boolean set, and tokens can only either exist or not.

3.3 Summary and Discussion

Petri nets are a powerful and convenient tool to model many kinds of dynamic systems. They allow for precise specification, simulation and analysis of systems by virtue of their rigorous mathematical formulation and convenient graphical representation. A vast number of extensions have been applied to them, of which one can employ as necessary to suit the application at hand.

PNs can be used as discrete controllers in industrial applications, where places and transitions can be associated with system components. The same concept can be easily applied to robotic tasks, as has been done in numerous works. The places of a PN can be associated with primitive movements and component actions, while transitions can be synchronized on various system events, such as specific sensor events or reaching primitive goal states. However, for application in PbD, the PN task model would have to be learned and constructed from demonstration. This entails that the demonstrations have to be segmented into primitives that can be associated with places. Once the PN is constructed, it can be executed as a discrete controller to perform the task.

Chapter 4

System Implementation

Acquiring Petri net models of tasks from demonstration requires algorithms to automatically construct and later execute them to reproduce the task. These algorithms cannot work in isolation, and require other software components, for example to segment demonstrations into streams of symbols. Consequently, a complete prototypical system was developed to evaluate the proposed approach. However, since the main focus of the thesis is learning PN task models, other components were implemented for the sole aim of making the system work. Therefore, simplistic and rudimentary designs were favored for these components, without much regard to performance or robustness. This also serves to test the ability of the PN approach to perform in a sub-optimal environment, and to compensate for inaccuracies and delays.

The idea of constructing a PN model of a process from observations is not new. For example, in the field known as Process Mining, a PN model of a process can be discovered from event logs, in what is known as process discovery [38, 39]. The resulting PN is a subset of PNs called workflow nets. Several algorithms have been developed to extract these nets [40]. However, even though event logs is a very similar concept to that of primitive symbol streams used in the task PN approach proposed in this chapter and workflow nets are very similar to task PNs, process mining algorithms are not suitable for discovering Petri net models of tasks that fulfill the goals of the thesis. There are two reasons for this. The first and the most important reason is that process mining approaches do not handle temporal data, and the resulting net does not include them. The second reason is that workflow nets model actions as transitions, while places represent conditions such as the completion of an action. This is contrary to the design philosophy of task PNs presented in this chapter, which is derived from control interpreted PNs.

This chapter presents the implementation details of the system. Although the system is designed to be of general applicability, certain components were implemented specifically for the hardware used. Therefore, a brief account of the hardware and software infrastructure will be given first. Afterwards, an overview of the system will be presented, followed by detailed descriptions of each of the system component.

4.1 Hardware and Infrastructure

The hardware used to perform tasks consists of a KUKA LWR4+ lightweight robotic arm [6], and a BarrettHand BH8-282 robotic hand [41] attached to its end as a tool as shown in Figure 4.1. The arm has seven degrees of freedom (DoF), and is controlled by a dedicated computer controller, called KUKA Robot Controller (KRC). Programs can be written for the arm on the KRC with a proprietary programming language called KUKA Robot Language (KRL). More importantly, to allow for more advanced control applications, the arm can also be controlled from an external computer over an Ethernet connection via a communication protocol based on User Datagram Protocol (UDP), called Fast Research Interface (FRI) [42]. The KRC has a built-in gravity compensation mode, which allows demonstrations to be performed kinesthetically.

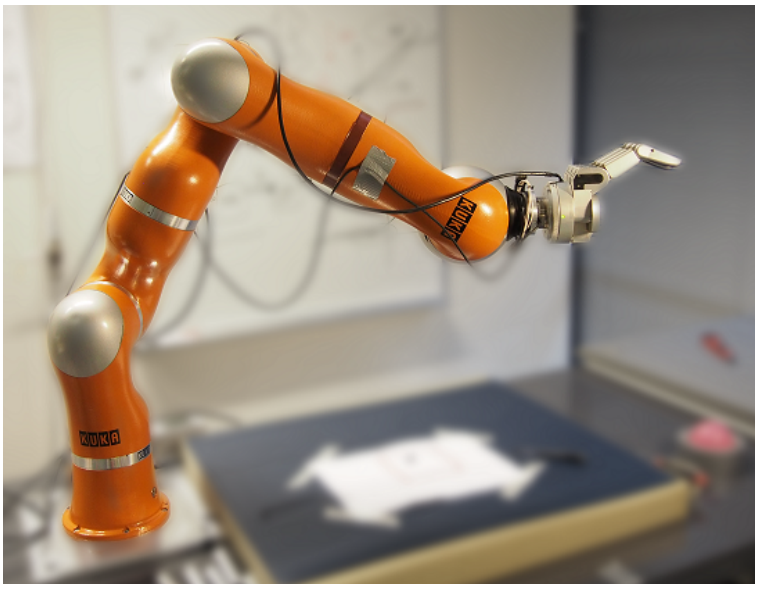


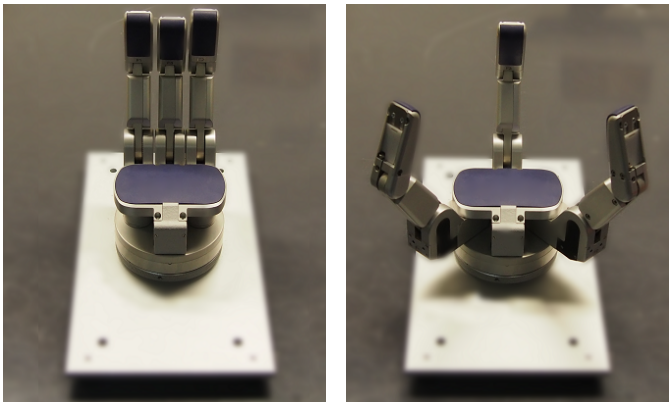
Figure 4.1: Robotic hardware used for the implementation of the system, showing the BarrettHand BH8-282 hand attached to the KUKA LWR4+ arm.

The hand has three fingers, each controlled by a single motor (Figure 4.2). The two outer fingers can rotate symmetrically around the palm; their angle is known as the spread, and is controlled by a fourth motor. The hand can be controlled from an external computer over a Controller Area Network (CAN) bus, which can send velocity, position, or torque commands to each puck, in addition to receiving feedback.

The external computer that controls the arm and the hand and on which the system is implemented runs the Linux operating system (Ubuntu 12.04) with a Xenomai real-time co-kernel [43]. The software components of the system were mainly implemented using ROS[1]. Communication with the KRC over FRI is handled by software components developed within the Orocos[2] framework

[1]ROS: Robot Operating System, is arguably the de facto meta-operating system for robot software development [44, 45].

[2]Orocos: Open Robot Control Software, is a collection of C++ libraries for advanced machine and robot control with real-time capabilities [46].



(a) (b)

Figure 4.2: BarrettHand BH8-282. (a) Spread angle of zero, and (b) of about 120 degrees.

for real-time support. A specific component that has been provided for the thesis, called the *KUKACommander*, abstracts the communication and provides higher-level functionalities, and is the main interface between the KRC and the rest of the system. Communication with the hand is handled using *Libbarrett*, an open-source C++ library available for controlling Barrett Technology products [47].

4.2 System Overview

Figure 4.3 shows a block diagram illustrating the main components of the system. The system currently only supports single demonstrations. First, a demonstration is performed using a multi-component robotic system composed of n components, and continuous trajectory data for each component is obtained. Each component trajectory is then fed into an associated segmenter object that implements a segmentation algorithm to segment it into symbolic representations of primitives. The segmentation algorithms can make use of a pre-learned library of primitives to identify primitives in trajectories. The segmentation stage results in n streams (or strings) of symbols for each component, which are then fed simultaneously into an algorithm that constructs a task Petri net (TPN). After the TPN is constructed, a PN controller executes it to obtain execution symbols, which contain information on what component should execute which primitive and for how long. The PN controller parses each execution symbol into a command for the appropriate component, and forwards it to the executor of that component. Each component is associated with an executor that handles communication with its dedicated controller, sending lower-level commands and receiving feedback. The executors also send feedback in turn to the PN controller, informing it that an execution has finished or stopped. The PN controller then forwards these events to the TPN it is executing, so that synchronized transitions can fire.

The general system structure makes no assumptions on the number of robotic

components used in demonstrations, nor on the method of providing demonstrations. However, some components need to be implemented specifically for the components being used, namely segmenters and executors. Therefore, the details of implementation given in the following sections are as applied specifically to the robotic hardware setup used.

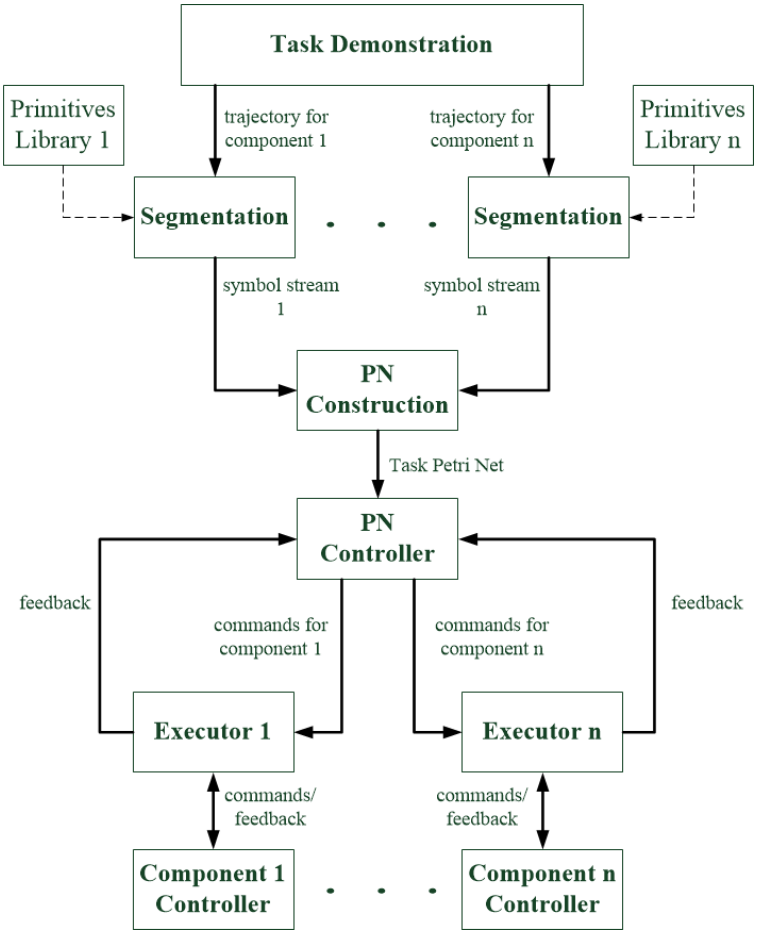


Figure 4.3: Proposed system overview.

4.3 Demonstration

Demonstrations were performed on the whole robotic system as if it was a single piece of hardware. Each of the three fingers in the hand and the spread is considered as a separate component. This gives maximum flexibility especially when using a rudimentary primitive framework, as well as being significantly simpler to segment. Together with the arm, this gives a maximum of 5 components. Not all of them have to be used however, and some can be left out of recordings.

To allow for kinesthetic demonstrations with the hand, a compliance controller had to be implemented for it since only the spread motor is backdriveable. This controller senses the torque on joints in each finger and sends a velocity command proportional to that torque to counter it. The reason that velocity

commands are used instead of torque commands is that the hand firmware inhibits torque commands to a finger when an external torque is applied to it. Furthermore, to facilitate demonstrations for the keyboard playing application, a special version of the compliance controller was implemented. In this version, the fingers move back to their initial position once the torque applied to them is lifted, as if they were attached with springs to this position.

The output of demonstrations is n trajectories for n components, each consisting of successive time-stamped data points. It is imperative that all trajectories share the same clock and start being recorded at the same time (even if it means recording a large stationary part), otherwise the temporal relationship between them will be lost. Trajectories are recorded in Cartesian task space for the arm, since this makes them much simpler to segment given the simple segmentation scheme described in Section 4.4. For the hand, the trajectory for each finger and the spread is just a one-dimensional joint position.

Demonstrations are recorded as ROS bag files, since they are easy to work with. More importantly, they also automatically provide time-stamping for data points according to a global clock. Moreover, ROS provides libraries for straightforward manipulation of bag files within programs.

4.4 Segmentation

The purpose of the segmentation stage is to transform continuous trajectories into discrete symbols that represent segments in these trajectories. A symbol is primarily a representation of a primitive-goal pair. It is implemented as a data structure that contains the id of the the component that originated it, the id of the movement primitive that represents the segment, and the goal state of the primitive (or endpoint of the segment). It also contains the start and end times of the movement, corresponding to the time-stamps of the first and last data points in the segment respectively. This also means that it implicitly contains information on the duration of the primitive.

The segmentation stage consists of two *segmenter* classes, one for the arm and one for all the hand components: the fingers and the spread. Both classes have a similar implementation, differing only in the number of trajectory dimensions and the value of their parameters. One object from the first class is constructed for the arm, and up to four objects of the second class for the hand components. Each segmenter object takes the continuous trajectory of its associated component as input, and outputs a symbol stream.

The segmentation scheme used is very simple, and is based on a vocabulary of just two primitives: the *move* primitive and the *stop* primitive. It is implemented as a FSM as shown in Figure 4.4. Before the actual segmentation starts, the algorithm goes through the whole trajectory and calculates the maximum Cartesian distance between any two successive data points. This distance is used to calculate a velocity threshold value (T_{go}) as a minute fraction of it, above which the component is considered moving. If no distance in the entire trajectory is above T_{go}, then the algorithm terminates and passes an empty symbol stream as output. Another threshold value (T_{stop}) is also calculated as a

slightly smaller fraction of the maximum distance, below which the component is considered stationary. This effectively creates hysteresis to avoid excessive noise in the segmentation output due to rapid switching between states.

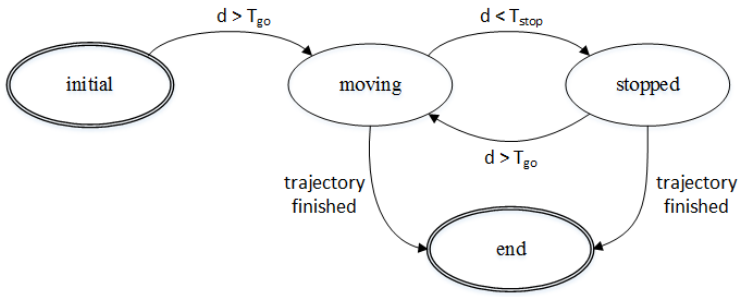


Figure 4.4: State machine of the segmentation algorithm. d is the distance between any two successeive data points, while T_{go} and T_{stop} are threshold parameters.

At the beginning of segmentation, the segmenter is in the *initial* state. As the algorithm goes through the data points, it calculates the distance between each two successive points. When this distance is greater than T_{go}, it switches to the *moving* state and stays there until the distance falls below T_{stop}, at which point it switches to *stopped* state. If the distance rises above T_{go} again, it goes back to the *moving* state and so forth, until all the data points in the trajectory have been processed, then it goes to the *end* state.

Every time the state switches between *moving* and *stopped*, a symbol is created and added to the output stream. When switching from the *moving* state to the *stopped* or *end* states, the primitive id is the *move* primitive. Its start time is the time-stamp of the last data point before the last switch to the *moving* state, its end time is the time-stamp of the last data point before the current switch, and its goal state is the state given by this data point. Similarly, When switching from the *stopped* state to the *moving* state, the primitive id is the *stop* primitive, and the symbol is called a *wait symbol*. Its start time is the time-stamp of the last data point before the last switch to the *stopped* state, and its end time is the time-stamp of the last data point before the current switch. It follows that any symbol has its end time identical to the start time of its successor. Note that no symbol is created when switching from the *initial* state to the *moving* state, nor when switching from *stopped* to *end*. This means that a stream always starts and ends with a non-wait symbol.

The symbol stream produced from this algorithm is quite noisy, with many superfluous symbols. This is mainly due to small unintentional movements made during the demonstration that lead to inconsistencies in readings hovering around the threshold values. To alleviate this over-reliance on the quality of the demonstration and decrease noise, two filtration stages are added in the signal path as shown in Figure 4.5.

In the first stage, all *wait* symbols whose duration is below a certain threshold value are discarded. Afterwards, the symbols just before and after a discarded symbol are merged if they have the same primitive id. When two symbols are merged, the resulting symbol keeps the start time of the earlier symbol, and

Figure 4.5: Filtration of the symbol stream.

the end time and goal state of the latter symbol. In the second stage, each non-wait symbol is examined, and its goals state is compared to that of the preceding non-wait symbol, skipping any wait symbols in between. If the distance between the two goal states is below a certain threshold, the symbol is discarded. As in the first stage, both symbols just before and after a discarded symbol are merged if they have the same primitive id.

The final symbol stream is relatively free of noise. The challenge is to adjust all the thresholds and parameters so that a noise-free symbol stream is obtained without significant loss of information. A side effect of the simple segmentation scheme used here is that all resulting movements are linear, and no curved movements can be reproduced. However, this suffices for the experiments discussed in Chapter 5.

4.5 Task Petri Nets

A *task Petri net* as currently implemented, is a timed, synchronized, and colored PN model of a task. It is also similar to the concept of control interpreted PN, except that places are associated with movement primitives, transitions can be timed as well as synchronized, and tokens are colored.

The structure of a TPN consists of a number of main branches equal to the number of components used in the task it models. The term 'branch' here is used to denote a series of mainly single-input, single-output places and transitions in succession. The relationship between branches and components is one-to-one, such that each branch correspond to exactly one component, and each component correspond to exactly one branch.

There is always one and only one start place in a TPN. This place is initialized with a single token in the initial marking, and spawns at least one branch. All branches can arise from either the start place directly, or another branch, depending on the time the corresponding component starts its first movement relative to other components.

4.5.1 Elements of a Task Petri Net

A TPN generally has the same elements as a colored PN. However, its elements are associated with those of the task it models. This sections lists the elements that constitute a TPN and their significance for a task model.

Places

A place in a TPN can have one of four types, each encoding a different type of information required to construct the model. The *ordinary* type, which is

by far the most common, is associated with a symbol from the symbol stream, such that a token in that place signifies that the associated symbol is active. The *count* type is used to count the number of repetitions in loops, and is initialized in the initial marking with a number of tokens equal to the number of repetitions of the loop. The *auto* type is auto-inserted in the construction phase to synchronize the beginning of a component branch with the other branches already executing. Finally, the *start* type is the start place of the whole TPN and is initialized with a single token in the initial marking.

Tokens and Color Sets

There are two color sets available in a TPN, and a place can be associated with one or both. The first color set is the set of positive real numbers $\mathbb{R}^+$, and the value of a token belonging to this color set signifies the duration of the execution symbol associated with that place. It follows that only ordinary-typed places can be associated with this color set. The second color set is the set of boolean values $\{true, false\}$, and tokens belonging to this color set can only either exist or not, exactly as in uncolored PNs. This color set can only be associated with ordinary places identified with wait symbols, or with count places. Boolean or uncolored tokens are used for synchronization between component branches when they exists in ordinary wait places, or for counting the number of loops when they exist in count places.

To simplify the implementation, the two color sets are reduced to just one, the set of non-negative real values $\{\mathbb{R}^+ \cup \{0\}\}$, while preserving the same modeling power. This is done by considering a token with a zero value as belonging to the boolean color set.

Transitions

Transitions in a TPN can either be synchronized or timed. Generally, output transitions of ordinary non-wait places (i.e. places associated with non-wait symbols) are synchronized on the event of the completion of the associated primitive. On the other hand, output transitions of ordinary wait places (i.e. places associated with wait symbols) are timed, and the delay is given by the duration of the wait symbol. All transitions are output transitions to ordinary-typed places, although they may have additional input places of other types.

Arcs and Arc Expressions

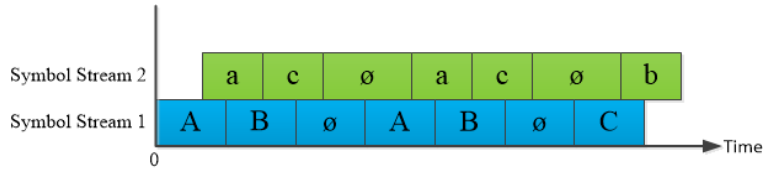
Since a TPN is colored, arcs have expressions. To simplify the implementation, expressions do not replace arc weights as in regular colored PNs, instead they complement them. An arc thus has both a weight that decides the number of tokens consumed or deposited, and an expression that operates on the real-valued tokens. This allows for the fundamental equation to be applied to calculate new markings.

Expressions are currently implemented as multipliers. A consumed real-valued token results in another real-valued token whose value is that of the consumed token multiplied by the multiplier. This is possible since, by design,

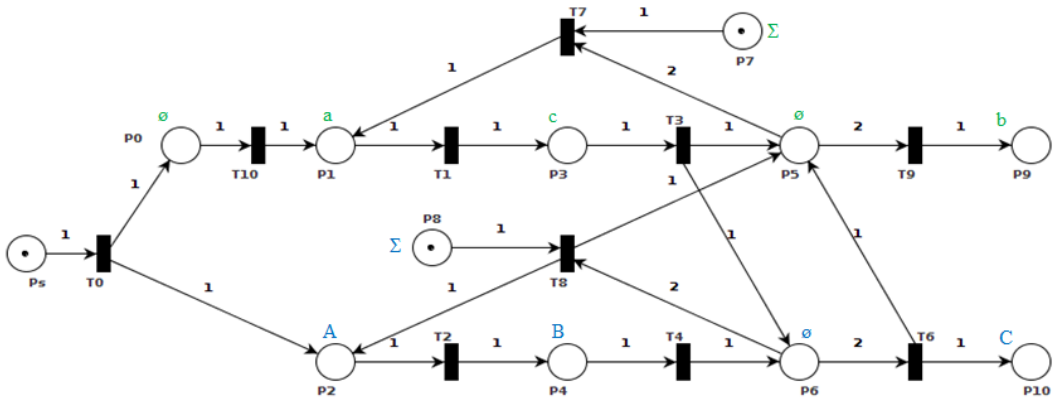
there can only be one colored (i.e. real-valued) token in a place at a time. Therefore, there is no confusion as to which token an expression applies to, even if the arc weight is greater than one. Moreover, only arcs from transitions to places have expressions, which is possible since the two color set of the TPN are effectively reduced to one as discussed earlier.

4.5.2 Modeling of Task Aspects

There are three main aspects of tasks that a TPN is required to model. The first of these is the task structure, which refers to the composition of a task in terms of its component movement primitives (or symbols) and their order. The second aspect is concurrency requirements and synchronization, and relates to which primitives of different components need to execute simultaneously. And lastly, the temporal aspect, refers to the time in which primitives are required to be executed and their duration.



(a) Two symbol streams from two components.



(b) The corresponding task Petri net in its initial marking. Σ denotes a count-typed place. Inscriptions for transition delays, synchronization events, arc expressions and token color are omitted for simplicity.

Figure 4.6: A symbol stream example and its corresponding task Petri net. a, b and c are symbols belonging to a certain component, while A, B and C are symbols belonging to another. $\varnothing$ is the wait symbol.

Task Structure

The task structure is the most basic and important aspect of tasks any modeling framework is expected to handle. Figure 4.6 shows a task example consisting of two symbol streams and the corresponding TPN model. Each symbol stream is

mapped to one branch of the TPN, reflecting the concurrent execution of both symbol streams. Moreover, the order of primitives is preserved in the sequence of places in each branch, reflecting the sequential execution of primitives in each symbol stream. A branch can spawn from the output transition of the start place, or from the output transition of a place in another branch, depending on the start time of the first symbol of the stream corresponding to this branch relative to the other streams.

Furthermore, repeating primitives in the task are modeled as loops in the TPN. For example, the fact that the sequence $(a, c, \varnothing)$ is repeated in the upper symbol stream in Figure 4.6(a), is modeled by the looping transition $T7$ in the TPN of Figure 4.6(b). The number of repetitions is given by the number of tokens in the accompanying place $P7$. Here, since $P7$ is initialized with a single token and has no input transitions, $T7$ can fire only once, reflecting the fact the repetition appears only once.

Concurrency and Synchronization

For tasks involving multiple robotic components moving at the same time, it is not sufficient to simply sequence the primitives. If the execution of a primitive for a component stalls for some reason, it will create a ripple effect delaying all others that succeed it. This might lead to loss of synchronization and concurrency since other components will not experience the same delay. For example, in Figure 4.6(a) there is a concurrency requirement that the last symbol b in the second stream be executed while C is executing in the first stream during reproduction. If the execution of any primitive preceding b in the same stream is delayed significantly, b might be executed after C has finished, thus failing the concurrency requirement. This effect can also happen if multiple primitives get delayed by even a small amount of time, as is often the case since robotic systems are hardly perfect.

TPNs as currently implemented can alleviate this problem. It is assumed that during a wait symbol, the component is waiting for other components to either start executing a certain primitive or finish executing one. It can be argued that this not the case for just wait symbols, and that a component may want to switch primitives based on what other components are doing without explicitly waiting. However, there's no way to identify such dynamic from single demonstrations. Therefore, if two non-wait symbols succeed each other without a wait symbol in between in a symbol stream, it is assumed that the latter symbol is just waiting for the preceding one to finish.

For instance, in Figure 4.6(a), the second a symbol in the second symbol stream has a wait symbol between it and the preceding c symbol, which would normally mean that it has to wait for the duration of the wait symbol after c has finished to start executing itself. However, symbol A from the first stream starts executing during that wait symbol, so now a is assumed to wait for A to start executing before it can start itself. This situation is modeled in Figure 4.6(b) by adding an arc from $T8$ to $P5$, and incrementing the weight of the arc from $P5$ to $T7$ by one. This is due to the fact that for A to start executing, $P2$ has to get a token, which will only happen when $T8$ fires. When $T8$ fires, it will now

also deposit a token in $P5$ thanks to the new arc. Now for a to start executing, $T7$ has to fire to deposit a token in $P1$, and this can only happen when $P5$ has two tokens since the arc between it and $T7$ has a weight of 2. Finally, $P5$ can only get two tokens when both $T8$ and $T3$ have fired, signifying that both c has finished executing and A has started.

Time

The problem of modeling the temporal aspect of tasks can be divided into two questions: *when* to start executing a primitive, and for *how long*. The time at which a primitive of a symbol is executed is decided by its preceding symbol. If a symbol is preceded by a non-wait symbol, then it executes immediately after the preceding symbol finishes. On the other hand if it is preceded by a wait symbol, then it waits for an amount of time given by the duration of this symbol before executing. In a TPN, timed transitions are used to decide when a primitive should be executed. Any place in the TPN associated with a wait symbol is followed by a timed transition whose delay is given by the duration of this wait symbol. Notice that the branch corresponding to any stream that starts later than other streams begins with a place associated with a wait symbol, and the same rule is applied to it.

The duration of execution of non-wait symbols is decided by the color (i.e. value) of the token inside its associated place. The arc expressions in the TPN reflect the relationship between the durations of any two successive symbols in a branch. For example if the duration of a symbol is half that of its predecessor, then the expression on the output arc of the transition between them is 0.5. This means that if a token in the preceding place that has a value of 1 is consumed, then the deposited token in the succeeding place will have a value of 0.5. It follows that the temporal scale of the entire task reproduction is decided by the value of the token in the start place in the initial marking.

It is worth noting at this point that given the current implementation, it is sufficient to obtain the required duration of the execution of primitives from the duration of the symbols themselves. While this is undoubtedly the simpler approach to take, however TPNs were implemented with general applications and future extensions in mind. For example it might be necessary at one point to include the history of the actual execution times of previous primitives to decide the duration of the next primitive. This can be either to rewrite arc expressions so as to adjust succeeding execution durations according to delays in previous executions, or perhaps to determine durations on-the-fly according to certain environmental conditions, or even nondeterministically according to some probability distribution. It might also be necessary to include other execution parameters in token colors. In these cases, using colored tokens seems to be the most appropriate and general method to propagate information through the net to fulfill these requirements.

4.6 Task Petri Net Construction

Before the construction of the TPN, all the symbol streams are bundled into one object, called a *stream bundle*. This object provides the necessary logical encapsulation to process the different streams simultaneously. Most importantly, it provides the method *getNextSymbol()* that can be called to return symbols in temporal order regardless of which component stream they belong to.

The process of building the TPN and later executing it is general and work regardless of the primitives framework used, so long as the resulting symbols implement a certain interface. It starts by passing a stream bundle object to the TPN class constructor, and consists of three stages as shown in Figure 4.7: naïve PN construction, concurrency enforcement, and folding.

Figure 4.7: Stages of TPN construction.

Naïve Petri Net Construction

The first stage in constructing a TPN model is constructing a naïve PN representation of the task to serve as the groundwork for further refinement. This PN consists of n branches corresponding to n symbol streams, each consisting of a series of places representing symbols in a stream. The algorithm (Alg. 4.1) starts by creating a start place and an output transition for it, which will be called the start transition. Afterwards, a place is created for the first symbol in the stream bundle as an output place to the start transition. For each subsequent symbol in the bundle for the same component, a new place is created, as well as a transition from the previous place to new one, creating the component branch in the process. Once the first symbol in a different stream is encountered, a leading wait place is created as an output place to the input transition of the last place created, thus creating a new branch. Another place is also created for the symbol in question, and a transition is created from the leading wait place to it. Subsequent symbols are treated normally as in the first branch. This process continues until all symbols in the bundle have been processed.

Whenever a place or transition is created, it is linked with the previous element in the same branch with an arc, which always has a weight of 1. Furthermore, each output arc of a transition has an expression equal to the duration of the symbol associated with the output place divided by that of the input place. Note that at this stage, all transitions have a single input place, whereas transitions that spawn branches may have more than one output place.

While constructing the PN, lists of all places, transitions and arcs created are kept. All elements in a list have an id that is used to consistently reference them in other elements. Places contain the ids of its input and output transition, while transitions contain the ids of its input and output places. Similarly, arcs contain the ids of its originating and terminating nodes. Throughout the

```
Require: stream bundle B
  add start place
  add start transition
  for i = 0 to i = B.symbolCount do
    s = B.getNextSymbol()
    if i = 0 then
      add place for s after start transition
    else
      if s is first symbol in a new stream then
        add leading wait place to create a new branch
        add arc from last transition added to new place
      end if
      add place for s in its branch
    end if
  end for
```

Algorithm 4.1: Constructing a naïve Petri net.

TPN construction process, the ids of places always have the same order as the temporal order of their associated symbols in the stream bundle. This facilitates the mapping between places and symbols, and helps tremendously for temporal comparisons required in certain operations as will be evident later.

Concurrency Enforcement

After a naïve PN is obtained, the next stage is to enforce concurrency between its branches. In this stage, an algorithm (Alg. 4.2) uses wait places (i.e. places associated with wait symbols) to either set transition delay times, or force the execution of succeeding symbols to wait for other symbols in other branches to start executing. The algorithm goes through all the places of the naïve PN to find wait places. For each wait place, there are two scenarios possible. If its temporally succeeding place is in the same branch, this means that the component just has to wait for the duration of the wait symbol. Consequently, the output transition of the wait place is set as a timed transition and its delay is set to the duration of the wait symbol. On the other hand, if the temporally succeeding place is in a different branch, this means that the place following the wait place in the same branch should wait for its temporally preceding place in whichever branch to be active. Consequently, an arc is created from the input transition of that temporally preceding place to the wait place. Furthermore, the output transition of the wait place is set as a timed transition, and its duration is given by the time difference between the start of the symbol to be waited for and the start of the symbol that is waiting.

Folding

Folding is the third and final stage in the construction of a TPN, in which repeating sequences in the net are replaced with loops. Since the temporal order

```
Require: naïve PN N
  P = getPlaceList(N)
  for i = P.start to P.end do
    if P(i) is a wait place then
      if P(i) and P(i + 1) are in the same branch then
        t = getOutputTransition(P(i))
        d = getSymbolDuration(P(i))
        t.delay = d
      else
        n = index of the place after P(i) in the same branch
        t = getInputTransition(P(n − 1))
        add arc from t to p
        t = getOutputTransition(P(i))
        d = getSymbolStartTime(P(n)) − getSymbolStartTime(P(n − 1))
        t.delay = d
      end if
    end if
  end for
```

Algorithm 4.2: Enforcing concurrency in a Petri net.

of symbols is reflected in the order of places, repeating sequences of places correspond directly to repetitions in symbols. A sequence is considered repeating if it appears again immediately after it ends with nothing in between. Furthermore, only repetitions across all branches are considered true repetitions and are consequently processed.

The algorithm used in this stage is similar in structure to that used in [23]. The algorithm starts by searching the entire TPN for repetitions. Once a repetition is found, all its constituent places are merged with their counterparts in later instances of the repetition, and all lists and structures are updated to reflect the changes. The process then starts again, and keeps running until no repetitions are found. For each repetition found, a new count-typed place is created as an input place to the transition between the first and last places of the repeating sequence. In the initial marking, this count place is initialized with a number of tokens equal to the number of instances of the repetition found.

When searching for repetitions, the algorithm (Alg. 4.3) operates on the list of places. It starts by looking for repeating sequences of length equals to half that of the entire list. If none are found, the length is decremented by one, and this process is repeated until the length is less than two places. This effectively favors longer repetitions over shorter ones. For a sequence of length l, the algorithm searches for repetition by comparing each place P_i in the list with place P_{i+l}. If the two places are equal, then P_{i+1} is compared with P_{i+l+1} and so forth until P_{i+l-1} is compared with P_{i+2l-1}. Any two places are considered equal if their associated symbols are equal, and any two symbols are considered equal if they have the same component and primitive ids and both their goal states and durations are roughly equal within a certain tolerance. When a pair

of places being compared are not equal, the starting i is incremented by one, and the process starts again. The process stops when $(i + 2l) > L$ where L is the length of the list, at which point the sequence length l is decremented and the process is restarted. Whenever two adjacent sequences match, the starting i in the search window is incremented by l to search for other instances of the sequence.

Require: PN N
  P = getPlaceList(N)
  $repetitionCount = 0$
  for $l = P.length/2$ **to** 2 **do**
    for $i = P.start$ **to** $P.end - 2l$ **do**
      $k = i$
      while $[P(k)$ **to** $P(k + l - 1)] = [P(k + l)$ **to** $P(k + 2l - 1)]$ **do**
        $repetitionCount++$
        $k = k + l$
      end while
    end for
  end for

Algorithm 4.3: Finding repetitions in a Petri Net

When merging two places, the one with the higher id is absorbed into the lower id one and gets marked for deletion, since a lower id corresponds to an earlier symbol. Each input or output transition of the absorbed place is compared to those of the absorbing place, if an equivalent is found then it gets marked for deletion, otherwise it becomes an input or output transition of the absorbing place, by having the ids of its input and output places changed accordingly. Two transitions are considered equivalent if all their input and output places are equivalent. All arcs in the arcs list get updated to reflect these changes. Finally, the deletion of all elements that were marked for deletion takes place when the lists get updated after each folding process.

4.7 Petri Net Controller

The main function of the PN controller is to act as an interface between the TPN and the executors. It executes the TPN, receives execution symbols from it, translates them into commands for executors to perform, and sends feedback to the TPN so that synchronized events can fire. Before the TPN is executed it has to be initialized. This prompts the TPN to use the lists of elements it stores to build the following: the incidence matrix; the expression matrix which is similar to the incidence matrix but holds arc expression values; the conflict matrix which is a square matrix with length equal to the number of transitions in the net, and holds information about conflicts between transitions; and finally the initial marking vector. At initialization, the TPN object is also passed a time seed value, which is the value of the token in the start place in the initial

marking that governs the execution speed of the entire reproduction, and a time step value, which decides the rate at which the TPN is to be updated.

To execute the TPN, the PN controller periodically calls the *advance()* method provided by the TPN object. This method updates the time delays on enabled timed transitions, and calculates a list of enabled transitions, from which it obtains the characteristic vector based on the timed transitions whose delay is over, and synchronized transitions whose associated events have been registered. The TPN advances the state by calculating the fundamental equation, and uses the expression matrix to calculate the value of newly deposited tokens. Furthermore, if two transitions appearing in the characteristic vector are in conflict (given by a non-zero element with the corresponding indices in the conflict matrix), a conflict resolution scheme is employed to decide which of them should fire. In the current implementation based on single demonstrations, a conflict can only arise between a looping transition and a loop-exit one, in which case the looping transition is always favored.

In each iteration, the controller also calls the method *getExecSymbols()* on the TPN that returns execution symbols based on the change in the marking. Each execution symbol contains a unique symbol id, a component id, a primitive id, and a duration. The controller parses this information and sends the appropriate commands to the respective executors. The controller then receives feedback that a command it has sent is completed, at which point it calls the method *SignalExecEnd()* on the TPN, and passes it the ids of the execution symbol that has completed since the last iteration as an argument. This allows the TPN to fire transitions synchronized on the events that these execution symbols have been completed.

4.8 Executors

Since the primitive framework used currently is rudimentary, consisting only of one simple movement primitive, the implementation of executors is straightforward. Each executor receives from the PN controller a goal state to reach, as well as the duration in which to do so. An executor then communicates with the dedicated control software of its component to send lower-level commands and receive feedback in order to reach the goal state in time. Currently, there are two executors: one for the arm, and one for the hand.

The arm executor receives the Cartesian goal state and the duration of the movement, and forwards them to the KUKACommander, which in turn forwards them to an external trajectory generator (Figure 4.8). With the aid of a kinematic model of the arm, this trajectory generator uses inverse kinematics to generate joint trajectories to achieve the goal state in time. It communicates directly with the KUKACommander, sending and receiving desired and measured joint states and other feedback signals. The generated trajectory has a trapezoidal velocity profile. Once the desired goal state is reached, the KUKACommander sends a signal to the executor, prompting the latter to signal the completion of an execution to the PN controller.

The hand executor runs four threads, each implementing a server to ser-

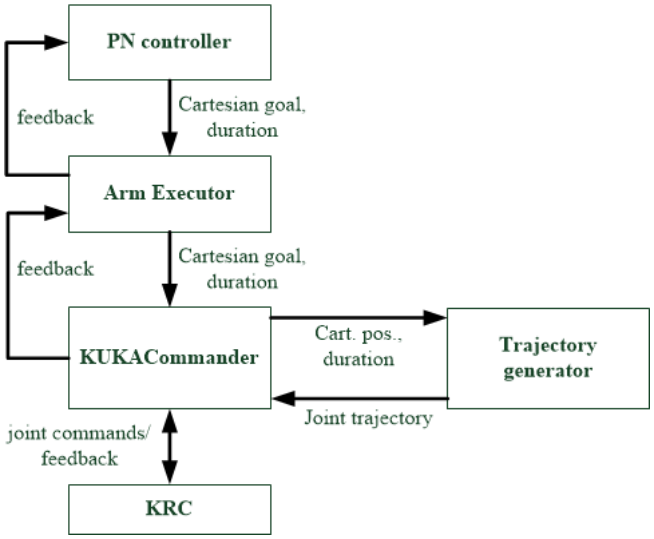


Figure 4.8: Signal path of arm commands.

vice execution commands sent to each of the four components comprising the hand: the three fingers and the spread. Therefore, from the PN controller's perspective it is seen as four separate executors (Figure 4.9). Each server receives a goal state and the duration of the movement, and translates them to velocity commands for the hand control software. The executor then collects these commands from all servers and sends them to the hand control software. A command is considered complete when the current position of a component is within a certain tolerance of its goal, at which point the executor signals the completion of the command to the PN controller.

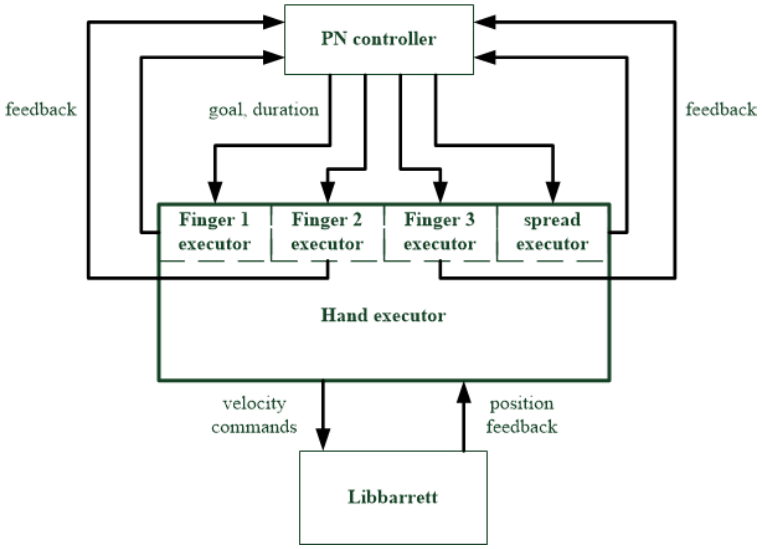


Figure 4.9: Signal path of hand commands.

4.9 Summary and Discussion

This chapter presented an overview of the system developed for the thesis, as well as implementation details of each of its components. The most important component of the system, and the main contribution of the thesis, is the construction of TPNs from demonstrations. TPNs model several aspects of tasks, such as the task structure, concurrency and synchronization requirements, and temporal data. A TPN can also be easily executed to reproduce the task.

A rudimentary primitive and segmentation framework was used, which allowed for fast prototyping of the system at the cost of imposing limitations on accuracy and the kinds movements that can be performed. Moreover, the system currently works on single demonstrations, therefore it is not capable of generalizing upon tasks. However, it was designed to be general enough to work regardless of the number or nature of robotic components used, or the primitive framework employed. It was also designed to provide a basis for further extensions and refinements, and some functionalities were implemented in anticipation of such extensions.

Chapter 5

Experiments

While the proposed approach based on Petri nets has been shown to be a theoretically viable solution, its feasibility can only be validated through concrete experimental results. Furthermore, it is required to test and evaluate the performance of the entire system, from task demonstration to reproduction. To this end, an experiment was designed and performed, in which the robot learns and reproduces a keyboard-playing task.

5.1 Overview

The purpose of the experiment is to evaluate the capability of the system to preserve temporal information and structure in tasks, and to enforce synchronization and concurrency requirements in several robotic components. Keyboard playing is a prime example of a task that places strict requirements on these aspects. The arm and each of the fingers have to be synchronized almost perfectly in order to hit the correct note (and only the correct note) in a very limited time window. Moreover, the duration of notes and silence in between have to be preserved for the reproduction to be considered successful.

For time-critical tasks, it is important to differentiate between two types of timing errors: delay and jitter. *Delay*, also called *latency*, is the difference between the time an event is supposed to occur, and the time it actually occurs. On the other hand, *jitter* is the variation of delay in successive events. Generally, lower jitter is more important than lower delay. For example, in keyboard playing if a note is delayed, the following note should still be played after the correct amount of time has passed, and not at the time it would have been played had the previous note not been delayed. Therefore, the relative timing between notes is more important than their absolute timing, or in other words, delay should be sacrificed to reduce jitter if need be.

Keyboard playing also offers a convenient way of systematically evaluating the performance of the task reproduction. A MIDI-enabled[1] musical keyboard can send MIDI signals of note on/off events, which can be recorded as MIDI

[1]MIDI: Musical Instrument Digital Interface, is a ubiquitous technical standard that defines a digital interface and protocol for communication between electronic musical instruments and computers [48].

tracks by a computer and analyzed. Midi tracks contain the sequence number and time-stamps of the events it recorded, among other information. Thus, by comparing different MIDI tracks, it is possible to compare the performance of the reproduction to that of the original demonstration with a temporal resolution in the order of milliseconds. This has the advantage of bypassing a non-trivial problem that would be present in most other task domains; the extraction of events on such temporal scale in other tasks would be significantly more difficult, assuming they can easily be defined in the first place.

5.2 Metrics

For quantitative evaluation of the performance of the reproduction, two metrics were used. The first metric is the *event time error* (ETE), and relates to the delay in events. For some reproduction of a demonstration containing n events, the ETE of the i-th event is defined as:

$$\text{ETE}_i = (t_i - t_0) - (t'_i - t'_0); \quad i = (0, 1, 2, \ldots, n-1), \tag{5.1}$$

where i is the event index starting from 0 for the zeroth event (the very first event), t_i is the time of the i-th event in the reproduction, and t'_i is the time of the i-th event in the original demonstration. Naturally, the ETE is zero for the zeroth event, since the latter only serves to synchronize the clocks of a demonstration and its reproduction.

The second and most important metric is the *inter-event time error* (IETE), and relates to jitter. The IETE of the i-th event is given by:

$$\begin{aligned} \text{IETE}_i &= \text{ETE}_i - \text{ETE}_{i-1} \\ &= (t_i - t_{i-1}) - (t'_i - t'_{i-1}); \quad i = (1, 2, \ldots, n-1). \end{aligned} \tag{5.2}$$

The IETE describes how well the temporal relationship between events is preserved in a demonstration, and is required to be as low as possible for all events. Since it is given by the difference in ETEs of two successive events, this entails that if the ETE of an event increases, the ETEs of all subsequent events must increase by the same amount.

5.3 Procedure

The robotic hardware used is as described in Section 4.1. The hand was mounted firmly on the arm, and the assembly was moved to a pre-programmed starting position in which the hand is in a horizontal position, with the fingers adjacent to each other and facing downwards as shown in Figure 5.1. A MIDI keyboard was placed underneath the hand, and connected via USB to a computer running a music sequencer that records MIDI tracks. The computer was supplied with an external midi interface that uses an ASIO[2] driver for low latency in MIDI signals.

[2]ASIO: Audio Streaming Input Output, is an extremely popular driver protocol for digital audio known for its low latency [49].

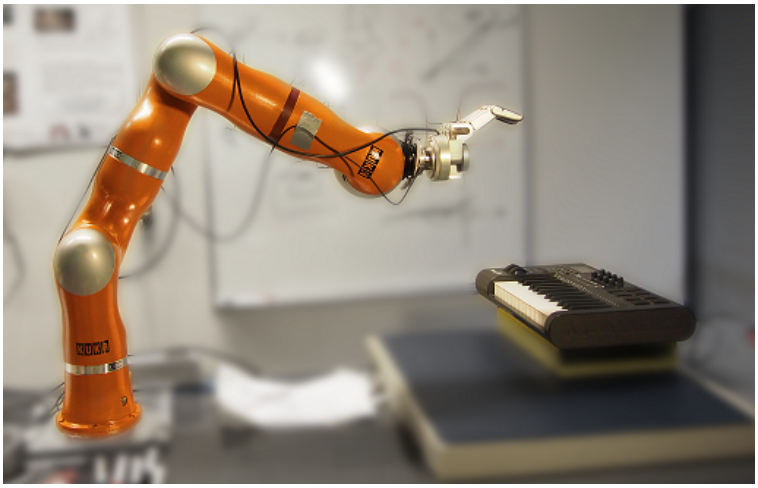


Figure 5.1: The setup used for the experiment, showing the hand attached to the arm, and the MIDI keyboard.

After moving the arm and the hand to the initial position, the demonstrator performed the demonstration kinesthetically (Figure 5.2). The demonstrated task was playing a small musical passage[3] on the keyboard consisting of fourteen notes across five keys. The demonstrator placed their hand on top of the robotic hand and aligned their fingers with those of the robotic hand. they then guided their arm with the help of their other hand, and used their fingers to push down the fingers of the robotic hand to hit the desired notes. The fingers move back to their initial position after the demonstrator lifts the torque they apply with their fingers. While the demonstration was being performed, the music sequencer recorded the MIDI track of the demonstration (Figure 5.3). The trajectories of both the hand and the arm were recorded as a bag file.

(a) (b)

Figure 5.2: Demonstration and reproduction of the keyboard-playing task. (a) the task being demonstrated kinesthetically, and (b) the robot reproducing the task.

The demonstration bag file was then segmented and the TPN constructed, initialized and executed. The TPN was initialized and executed three times at different speeds. The time seed value was first initialized with a value of 1 for

[3]For this particular experiment, a small passage from Beethoven's ninth symphony was played.

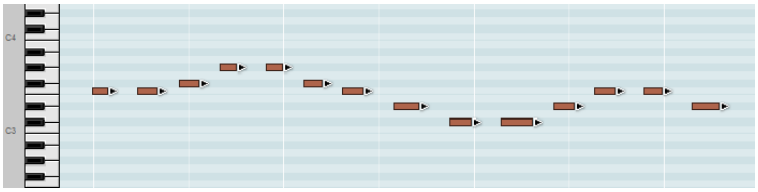


Figure 5.3: MIDI track of the original demonstration.

execution at the original demonstration speed, then a value of 0.5 for execution at double speed, and finally a value of 0.33 for execution at triple speed. MIDI tracks of the reproduction at each execution speed were recorded. Furthermore, the delay in the execution of each execution symbol for all components in each reproduction were calculated and recorded for further analysis. This execution delay is the difference between the duration in which a primitive was executed, and the duration commanded to the corresponding executor. In all reproductions, the TPN was updated at a rate of 300 hz.

In addition to reproductions by executing the TPN, reproductions by simple replay of the demonstrated trajectory were also recorded at original, double, and triple speeds. For the arm, this was done by sending the Cartesian data points in the original trajectory as setpoints over the FRI to the internal Cartesian stiffness controller in the KRC. The control law is given by [42]:

$$\tau_{\text{cmd}} = J^T(K_c(x_{\text{FRI}} - x_{\text{msr}}) + F_{\text{FRI}}) + D(d_c) + f_{\text{dynamics}}(q, \dot{q}, \ddot{q}) \tag{5.3}$$

where τ_{cmd} is the torque commands to the joints, J^T is the transposed Jacobian matrix, K_c is the stiffness parameter, x_{FRI} is the Cartesian position setpoint, x_{msr} is the measured Cartresian position, F_{FRI} is a the desired Cartesian force/-torque, $D(d_c)$ is the damping term given by the damping coefficient d_c, and $f_{\text{dynamics}}(q, \dot{q}, \ddot{q})$ is the dynamic forces compensation term. The stiffness and damping parameters can be changed by the user to emulate a virtual spring. Replaying the hand trajectory was done by sending position setpoints for all the motors to internal proportional-integral-derivative controllers (PID controllers) in the hand firmware.

5.4 Results and Analysis

For analysis of the results, the MIDI file containing all recorded MIDI tracks was parsed to extract event times in each track. In total there are seven tracks in a file: one for the original demonstration, three for TPN reproductions at the three speeds, and three for the trajectory replay at the three speeds. Since the musical passage played contained fourteen notes, each MIDI track contained twenty-eight events. Finally, the ETEs and IETEs were calculated for all events.

Figure 5.4 shows the result of the reproduction at the original demonstration speed. Here, the trajectory replay was performed using a high stiffness value of 4000 N/m and a damping of 0.7 N.s/m for the arm. The mean of the execution delay for the arm was $\mu_{\text{ED}} = 15.6$ ms with a standard deviation $\sigma_{\text{ED}} = 4.1$ ms, while for the hand components $\mu_{\text{ED}} = -6.2$ ms with $\sigma_{\text{ED}} = 10$ ms.

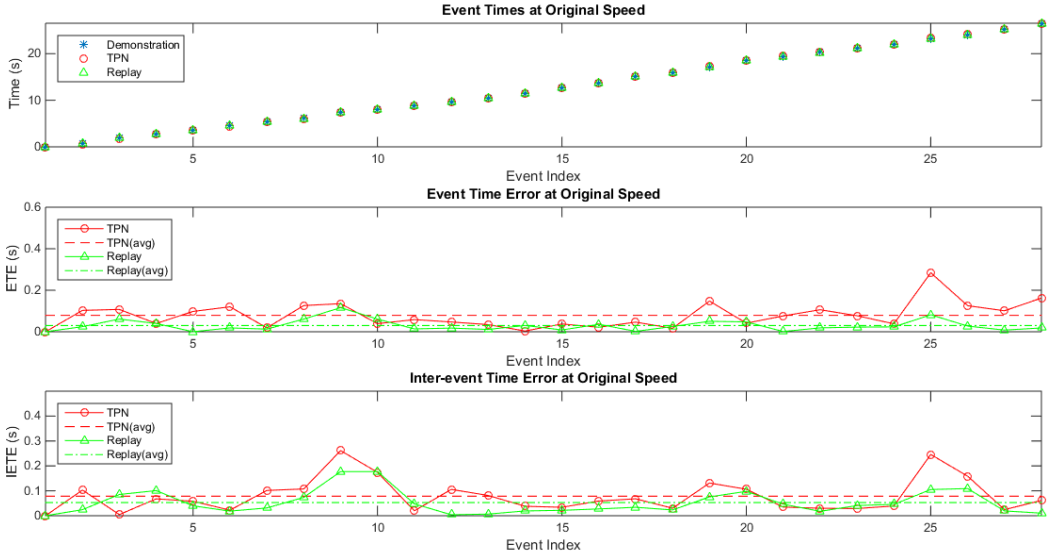


Figure 5.4: Results of reproduction at normal speed with trajectory replay at high stiffness.

The reproduction by the TPN generally compensated for delays in execution and kept the IETEs low. However, the joint trajectories generated for the arm by the trajectory generator had a trapezoidal velocity profile. Consequently, at such a low speed the fingers did not hit the keys with sufficient force as the arm slowed down as it approached its goal position. This resulted in a delay between the time a key is pressed and the time the arm reached its goal that was not present in the demonstration, as exemplified by peaks in the ETEs and IETEs of both the replay and the TPN reproduction at events 9, 19 and 25 when the pressing force was fully supplied by the arm. This delay could not be inferred from the overall execution delay of the command, and thus could not be fully and immediately corrected by the TPN. Furthermore, the average arm execution delay was positive and large, while that of the hand was negative with large σ, which led to complex patterns of interaction that resulted in unpredictable delays. Nevertheless, the IETEs were stabilized by the TPN, and the TPN reproduction was still comparable to the original demonstration and the trajectory replay. For the TPN, the mean IETE was $\mu_{\text{IETE}} = 78.4$ ms and the standard deviation $\sigma_{\text{IETE}} = 66.7$ ms, while for the replay $\mu_{\text{IETE}} = 52.8$ ms and $\sigma_{\text{IETE}} = 47.7$ ms.

This extra delay problem was alleviated to some extent in the reproduction at double speed as shown in Figure 5.5, although minor spikes can still be seen at events 9, 19 and 25. The original demonstration event times were halved to obtain the demonstration at double speed, which is used to calculate the metrics for the reproduction. At this speed, for the arm execution delay $\mu_{\text{ED}} = 12.1$ ms and $\sigma_{\text{ED}} = 3.8$ ms, while for the hand $\mu_{\text{ED}} = 3.7$ ms and $\sigma_{\text{ED}} = 2.8$ ms. Here, since the average execution delay of both the arm and the hand was positive, and TPN always compensated for delays, the ETE of the TPN is almost always rising. This is required to keep the IETE low as discussed in Section 5.1. Although the average ETE of the TPN is significantly higher

than that of the trajectory replay, the IETEs for both are comparable. For the TPN, $\mu_{\text{IETE}} = 39.5$ ms and $\sigma_{\text{IETE}} = 28$ ms, while for the trajectory replay $\mu_{\text{IETE}} = 25.3$ ms and $\sigma_{\text{IETE}} = 27.7$ ms. It is worth noting that the performance at double speed is better that at the original speed, and although this might seem counter-intuitive, it can be attributed to the fact that moving at greater speeds means delivering sufficient force to press the keys hard enough, so that events are registered more closely to the appropriate time. This is also reflected in the fact that the execution delays for both the arm and the hand were lower.

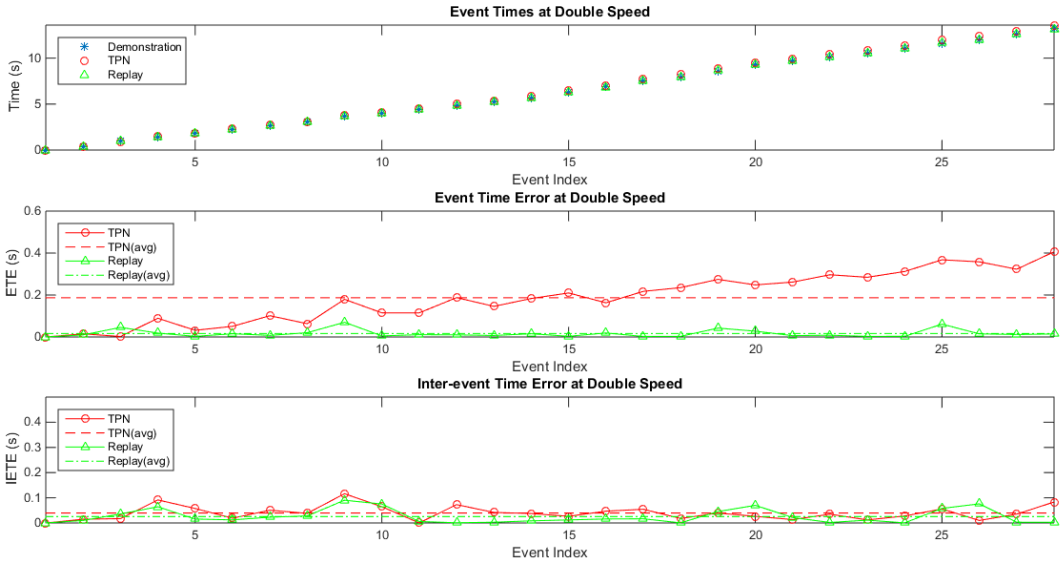


Figure 5.5: Results of reproduction at double speed with trajectory replay at high stiffness.

Reproduction at triple speed shows a similar trend as shown in Figure 5.6. At such high speed, execution delay deteriorated significantly for the arm at $\mu_{\text{ED}} = 33$ ms and $\sigma_{\text{ED}} = 57.3$ ms, and slightly for the hand at $\mu_{\text{ED}} = 10$ ms and $\sigma_{\text{ED}} = 5.1$ ms. Furthermore, the delay in the trajectory replay becomes obvious as the arm struggled to follow the trajectory in time. For the TPN reproduction, $\mu_{\text{IETE}} = 28.4$ ms and $\sigma_{\text{IETE}} = 21.7$ ms, while for the trajectory replay $\mu_{\text{IETE}} = 24$ ms and $\sigma_{\text{IETE}} = 25.1$ ms. Although the mean IETE for the TPN is slightly higher than that of the trajectory replay, the standard deviation is lower, signifying more consistency in the TPN reproduction.

In the results discussed so far, the trajectory replay was almost perfect as a result of the very high stiffness value used. If significant delay was introduced in the trajectory replay, for example by setting the stiffness to a lower value, the advantage of the TPN reproduction would become clear. Figure 5.7 shows the results of the reproduction with the stiffness for the trajectory replay set to 1000 N/m. At double and triple speeds, it can be seen that the IETE for the TPN reproduction always plummets after it peaks, as the TPN senses the delay and corrects it for the succeeding events. This is in contrast to the IETE of the trajectory replay, where double peaks are not uncommon. The performance of the TPN reproduction was consistently superior to that of the trajectory replay.

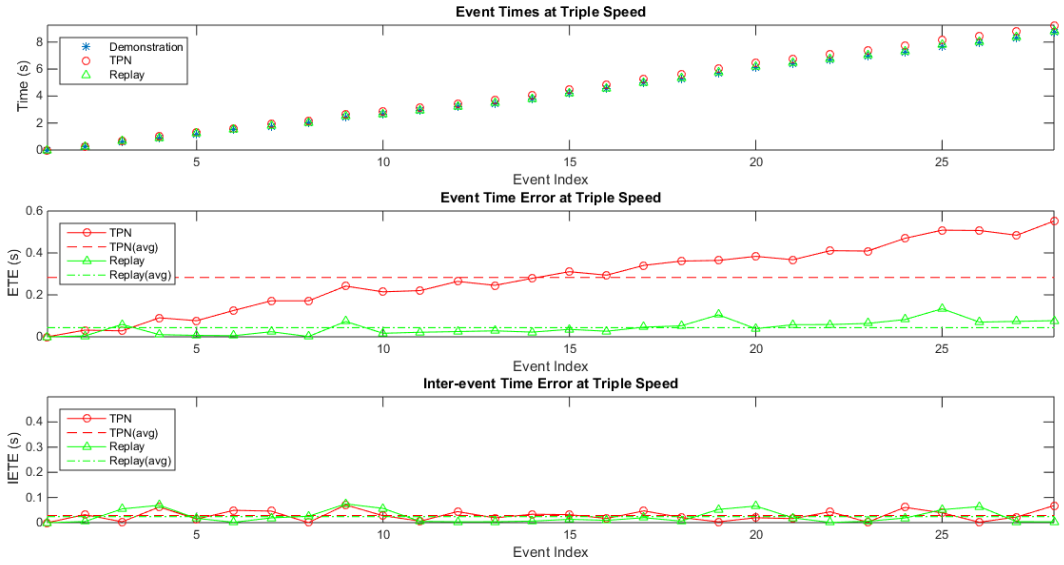


Figure 5.6: Results of reproduction at triple speed with trajectory replay at high stiffness.

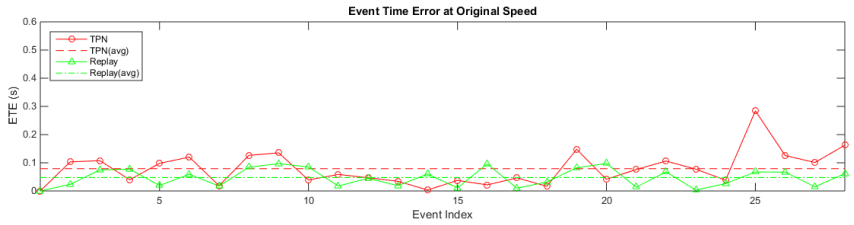
The statistics of the IETEs for both the TPN reproduction and the trajectory replay are given in Table 5.1 for this case, as well as for the high stiffness one.

5.5 Summary and Discussion

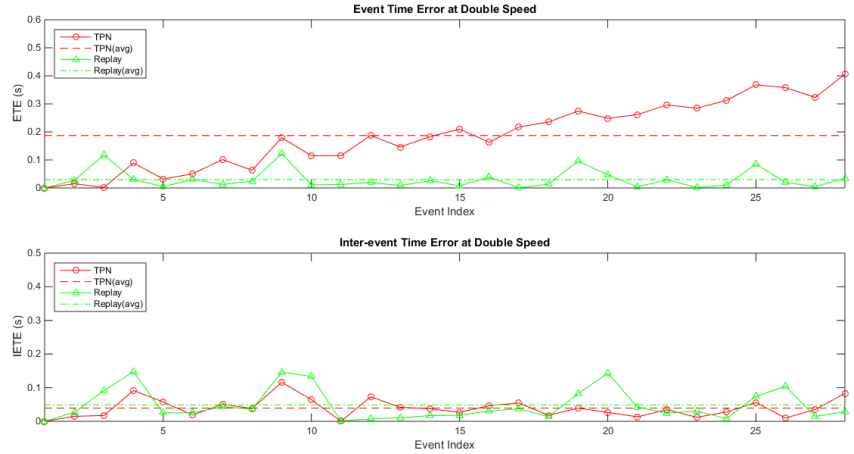
This chapter detailed the experiment performed to validate the Petri net approach to modeling tasks in PbD and evaluate its performance. In the experiment, a small musical passage was demonstrated kinesthetically on a MIDI keyboard. The system reproduced the task reasonably well, preserving the task structure as well as fulfilling timing and concurrency requirements, as well as compensating for the variability in execution delay times. However, The rudimentary primitive and segmentation framework used in the system imposed limitations on the quality of reproductions and introduced accuracy and timing errors that were sometimes impossible to compensate for. At low speeds,

Table 5.1: Mean and standard deviation values for the IETE and execution delay at different reproduction speeds. R-HS and R-LS are the trajectory replay at high and low stiffness respectively. All values are in milliseconds.

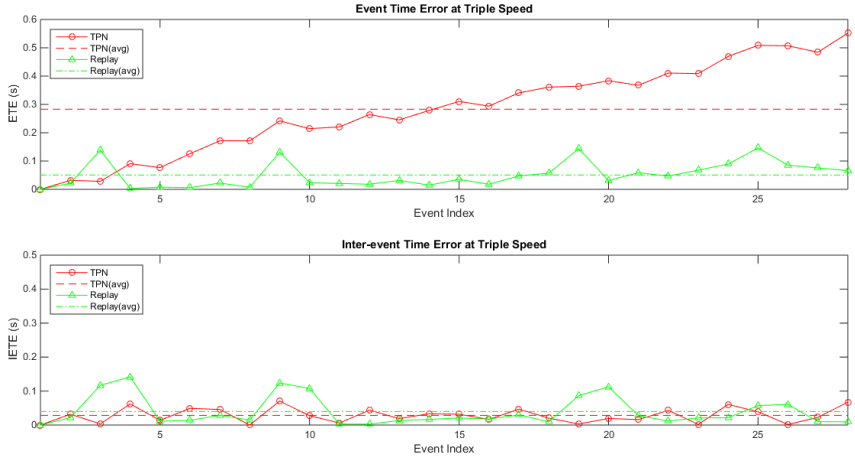
	Normal Speed			Double Speed			Triple Speed		
	TPN	R-HS	R-LS	TPN	R-HS	R-LS	TPN	R-HS	R-LS
μ_{IETE}	78.4	52.8	77.8	39.5	25.3	48.7	28.4	24	39.7
σ_{IETE}	66.7	47.7	50.7	28	27.7	46.9	21.7	25.1	42.9
Arm μ_{ED}	15.6	-	-	12.1	-	-	33	-	-
Arm σ_{ED}	4.1	-	-	3.8	-	-	57.3	-	-
Hand μ_{ED}	-6.2	-	-	3.7	-	-	10	-	-
Hand σ_{ED}	10	-	-	2.8	-	-	5.1	-	-



(a)



(b)



(c)

Figure 5.7: Results of the reproduction with trajectory replay at low stiffness. (a) at normal speed, (b) at double speed, and (c) at triple speed.

when these errors are more pronounced, the trajectory replay with high stiffness performed significantly better than TPN reproductions. This is expected since the trajectory replay was almost perfect, and no modeling framework based on single demonstrations can result in a better performance than a perfect replay of the demonstration. At high speeds however, the TPN performance was comparable to trajectory replay. Furthermore, when the stiffness for the trajectory replay is reduced to simulate delays, the TPN reproduction offered noticeably better performance.

The delays due to insufficient force at low speeds can be eliminated by using a more advanced primitive framework such as DMP. This framework would result in better reproduction of individual primitives, and consequently better performance of the whole system. Furthermore, the inclusion of force requirements in such framework as in [4, 5] would ensure keys are hit with sufficient force at low speeds.

The main advantage of obtaining a TPN model of the task from a single demonstration and using it for task reproduction is not merely improved performance over trajectory replay. Acquiring a concise model of a task allows for generalization by comparing the multiple models obtained from multiple demonstrations of the task, which is practically impossible to do just from trajectories. The possibility of incorporating multiple demonstrations as well as other extensions to the system are discussed in the following chapter.

Chapter 6

Conclusions

The goal of this thesis has been to develop a modeling framework in programming by demonstration that can fulfill two requirements: to be able to model time-critical tasks and preserve temporal relationships; and to be able to learn tasks performed by robots consisting of heterogeneous parts and model concurrent movements and synchronization requirements between them. This framework was to be theoretically developed and experimentally verified as a viable solution, providing a basis for further development and extensions.

A review of the prevalent approaches to PbD in the literature was presented, and it was concluded that none of them was adequate to fulfill the aforementioned requirements. This prompted the development of a new approach based on Petri nets to meet the goals of the thesis. Inspired by the use of control interpreted Petri nets as discrete controllers, the notion of task Petri nets was developed and introduced as models of tasks in PbD.

The concept of learning a complicated task as a sequence of movement primitives and using algorithms to construct a graph model of the task from segmented demonstrations has been introduced in the literature as in [23]. However, although the approach in [23] is similar in essence to the one proposed in this thesis, the former does not model temporal data, nor allows multiple primitives to be active simultaneously. Moreover, simply extending the algorithms employed there does not suffice, since the modeling framework used is based on state machines which by nature are not conducive to such extensions. The use of PN-based models on the other hand makes it possible to implement these aspects. Although PNs have been used before in a PbD setting to learn tasks as in [29], no attempt was made to use them to synchronize multiple movements or model temporal data.

In the course of the thesis work, a complete system was designed and implemented in which TPNs are learned and constructed from demonstrations, and later executed to reproduce tasks. The most important component of the system and the main contribution of the thesis is the development of algorithms to construct TPNs from segmented demonstration data. Due to the limited time available for development, simplistic designs for other components such as segmenters and executors were chosen at the cost of accuracy and robustness. The focus was on making the whole system operational, and to allow for testing the process of TPN construction and execution.

To validate the proposed approach and evaluate the performance of the developed system, an experiment was performed in which a robot learns a task from demonstration and reproduces it. The task was playing a short musical passage on a keyboard. Experimental results and analysis of the reproduction showed that the system was able to learn the task structure as well as timing and synchronization requirements. Delays in the execution of movement primitives were generally compensated for, and the overall task was reproduced in a timely manner. Nevertheless, the simplistic and rudimentary primitive and segmentation framework employed still adversely affected the performance and imposed limitations on the kind of movements that could be reproduced. However, an important advantage of the proposed system is that the primitive and segmentation framework is decoupled from the TPN construction process. This allows for straightforward replacement of that part of the system with a more advanced one, or even simultaneous use of different frameworks for different components.

Since the current implementation of the system mainly serves as a prototype to validate the approach, it still has many limitations. For example, the system is currently based on single demonstrations, and is thus incapable of generalization which is a crucial aspect of PbD. Furthermore, the system does not use feedback from the environment and does not possess any context for the action it performs. This means that the movements are performed blindly. For instance, in the keyboard-playing task if the keyboard is shifted slightly sideways or removed altogether, the robot would still make the exact same movements, resulting in wrong notes or in nothing at all. This also means that the system is incapable of learning tasks that cannot be fully modeled as just a combination of movement primitives, and which would require perception of the environment. Another limitation is the inability to synchronize movements of different parts except at the onset of the execution of a primitive. A consequence of this is that synchronization might be lost when executing a primitive with a long duration. However, most of these limitations can be eliminated by extending the system as discussed in the following section.

Future Work

The purpose of the developed system is not merely to provide better performance from single demonstrations than trajectory playback. It is meant to serve as the groundwork for further extensions and refinement. There are numerous ways to extend the current system in order to realize the full potential of the approach and firmly establish it as an effective solution in PbD, some of which are presented in this section.

Primitives and Segmentation

As a prerequisite to further extensions, a proper primitive framework will have to be used. Such framework would allow relatively complex movements to be learned and perfected before the demonstration of the actual task, as well as faithful and accurate reproduction of individual primitives regardless of the goal

state. This would dramatically enhance the performance of the whole system, and the ability to learn more complex component movements would in turn allow for learning more complex tasks. It would also be advantageous for the primitive framework to be able to learn force/torque requirements, which would allow learning in-contact tasks as in [4, 5].

Furthermore, this primitive framework should be coupled with a robust segmentation scheme to be useful. The output of the segmentation stage should be accurate and consistent for slightly different demonstrations. This is important for detecting loops, since a repeating movement is extremely unlikely to be repeated in exactly the same manner. For a similar reason, robust segmentation would also be a requirement for incorporating multiple demonstrations. In addition, a probabilistic segmentation output would be useful for multiple demonstrations, and would result in better generalization. DMP certainly seems a good candidate as a primitive framework that fulfills these requirements as discussed in Section 2.3.2.

Generalization from Multiple Demonstrations

Once a proper primitive and segmentation framework is employed, perhaps the most important extension would be generalizing from multiple demonstrations. Generalization is crucial to learning tasks, and no PbD approach can be considered successful without generalizing upon demonstrations.

Generalizing from multiple demonstrations can be achieved by obtaining a PN model of each demonstration and then combining them together. A new model of a new demonstration would be compared to an existing one, and branches would be introduced at the point the two models differ, signifying alternative paths as in other graph-based methods in the literature. The probabilistic output of segmentation can be used to determine the confidence in an alternate path in a task, such that the decision to include a certain path in the final model depends on the support the path has, expressed in terms of the probability of the primitives comprising the path as output by segmentation, and the number of occurrences of the path relative to the number of available demonstrations[1], similar to how SCFG rule probabilites are determined in [27, 28]. Alternative paths with support above a certain value can be added to the net, and the choice of which path to follow can be made arbitrarily, or in a probabilistic fashion depending on the value of the support.

In addition to task structure, other parameters of TPNs such as arc expressions, number of repetitions in loops, and transition delays can also be generalized from multiple demonstration. In the simplest case, the average of such values over all demonstrations can be computed and used in the final model. Alternatively, a window of permissible values can be calculated, and the selection of specific values that optimize a particular metric can be made at execution time.

[1] This is similar to the concept of support used in data mining [50], but adjusted by the probability of the segmentation output.

Synchronization and Timing

Transition firing, which decides when switching between primitives occurs, is currently wholly dependent either on time or on primitive completion events. This means that for a primitive to start executing, the system has to either wait for a preset duration of time, or for the preceding primitive to finish executing. This adversely affects the flexibility of the system in three ways. First, this dependence limits the set of tasks the approach can effectively model, since some tasks may require switching primitives before a currently executing one reaches its goal state. This can be as a reaction to certain environmental events, or due to the nature of the segmentation output in which symbols can overlap. In fact, for skillful reproduction of a sequence of movement primitives, the primitives are often required to be concatenated or to spatially and temporally overlap at some point, in what is known as co-articulation [51]. Second, for some tasks, the number of repetitions in a loop can vary depending on some change in the environment. For example, tightening a screw requires repeating a certain sequence of movements not for a preset number of times, but until it can no longer be turned. Therefore, in order to exit loops, it can be required to synchronize loop-exit transitions on certain events that can be extracted from sensor data, instead of relying on looping transitions to no longer be enabled. Third, implementing generalization from multiple demonstrations requires a mechanism to select which of the alternative paths to follow. In some tasks, such decision can be traced to environmental events, in which case it would be required to extract these events to synchronize alternate transitions on them.

Furthermore, synchronization between primitives is currently only done by exploiting wait symbols. This can be problematic since if the segmentation results in a sequence of overlapping symbols for some component (i.e. with no wait-symbols in between), this might lead to loss of synchronization with other components. It might also be necessary to execute a primitive in precisely a certain point during the execution of another primitive of another component in order to successfully perform the task. In that case, a mechanism would have to be found that synchronizes the two primitives correctly.

These issues can be addressed by associating transitions with classifiers, similar to the approach employed in [23]. These classifiers would be trained using sensor features in labeled demonstration data to extract events on which transitions would be synchronized. For any synchronized transition, its associated classifier would be trained using sensor features from the component associated with its branch. On the other hand, for timed transitions, classifiers would be trained using features belonging to other components which are executing at the time of firing. This allows the model to capture synchronization requirements in tasks without the need for wait-symbols. The events produced by classifiers can decide when to switch between primitives in a sequence, when to start executing a primitive after waiting, when to exit loops, and which path of execution to select for a component to select if there are alternatives as a result of combining multiple nets.

Using classifiers to switch between primitives means that some timed transitions would have to be synchronized instead. It can be difficult to learn whether

a primitive should execute after a certain duration of time, or based on sensor features of another component active at the same time (i.e. whether a transition is timed or synchronized). However, it would be possible to employ some sort of Bayesian framework to make such decisions. For example, based on the first demonstration, an assumption is made that a certain transition is timed. As more demonstrations become available, the system looks for features in other components at the time of firing that are consistent across multiple demonstrations, increasing or decreasing its belief that the transition is synchronized. The variations in wait times across demonstrations can also be exploited to make the decision. Alternatively, it is also possible to make all timed transitions synchronized, and design classifiers to process temporal data in addition to sensor features.

Multimodality

The events on which transitions may be synchronized need not arise only from trajectories. Other sources of information in other modalities such as tactility, vision and sound can be used as well. In fact, for some tasks that require interaction with a dynamic environment, it is not sufficient to rely on trajectories alone, and learning the task correctly depends on critical information in other modalities. Depending on the nature of the modality and the task, image and audio processing or classification techniques can be used to extract relevant events.

Information Propagation

The mechanism to propagate information through the TPN is already implemented as token colors. Currently, the information encoded in token colors dictate the nominal duration of primitives to be executed. However, at execution time of the TPN, the actual duration of an executing primitive can vary, for example when delay is encountered, or when a transition fires prematurely (i.e. before the goal state is reached) as a result of an event raised by a classifier. It might be desired for certain applications to adjust the duration of succeeding primitives accordingly in order to maintain a uniform execution speed. In that case, the color of a token in an active place can be altered before its output transition fires, so that this information is passed on to succeeding places.

It is also possible for tokens to carry other types of information that can modulate the execution of primitives if necessary. There is no assumption made on the amount or nature of information that can be encoded as token colors. However, as token colors become more complex, so will arc expressions, and the challenge would be to find a way to efficiently encode and learn expressions from demonstrations.

Hierarchical Petri Nets

Petri Net models of large systems can be divided into interrelated subnets for more tractable modeling, in what is known as hierarchical Petri nets [52]. The same concept can also be applied to TPNs. This will effectively introduce

more layers of hierarchy into the task model to accommodate more complex tasks. Places in higher-level nets can be expanded into lower-level ones, and nets of both levels would have a similar structure to TPNs. As an example, in the keyboard-playing task of the experiment in Chapter 5, a high-level TPN can be introduced in which places are associated with actions that result in playing notes. These places can be expanded into lower-level TPNs that have been learned beforehand, each of which instructs the robot to execute a series of primitives to play a certain note. The introduction of such hierarchy on the symbolic level could lead to faster and more efficient learning of complex tasks as well as better generalization, in a similar manner to how the inherent hierarchy in symbolic approaches does.

References

[1] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[2] Aude Billard, Sylvain Calinon, Rüdiger Dillmann, and Stefan Schaal. Robot Programming by Demonstration. In Bruno Siciliano and Oussama Khatib, editors, *Handbook of Robotics*, chapter 59, pages 1371–1394. Springer, New York, NY, 2008.

[3] Chrystopher L Nehaniv and Kerstin Dautenhahn. *Imitation in animals and artifacts*. MIT Press, 2002.

[4] Alberto Montebelli, Franz Steinmetz, and Ville Kyrki. On Handing Down Our Tools to Robots : Single-Phase Kinesthetic Teaching for Dynamic In-Contact Tasks. In *2015 IEEE International Conference on Robotics and Automation*, 2015.

[5] Franz Steinmetz, Alberto Montebelli, and Ville Kyrki. Simultaneous Kinesthetic Teaching of Positional and Force Requirements for Sequential In-Contact Tasks. *Submitted to Humanoids 2015*.

[6] Rainer Bischoff, Johannes Kurth, Günter Schreiber, Ralf Koeppe, Alin Albu-Schäffer, Der Beyer, Oliver Eiberger, Sami Haddadin, Andreas Stemmer, Gerhard Grunwald, and Kuka Roboter Gmbh. The KUKA-DLR Lightweight Robot arm – a new reference platform for robotics research and manufacturing Summary / Abstract Stages of research and product de- velopment. *Joint 41th Internatiional Symposium on Robotics and 6th German Conference on Robotics*, pages 741–748, 2010.

[7] Aleš Ude. Trajectory generation from noisy positions of object features for teaching robot paths. *Robotics and Autonomous Systems*, 11(March 1991):113–127, 1993.

[8] Nathan Delson and Harry West. Robot programming by human demonstration: adaptation and inconsistency in constrained motion. In *International Conference on Robotics and Automation*, pages 30–36, 1996.

[9] J. Aleotti and S. Caselli. Robust trajectory learning and approximation for robot programming by demonstration. *Robotics and Autonomous Systems*, 54:409–413, 2006.

[10] Stefan Schaal, Christopher G Atkeson, and Sethu Vijayakumar. Scalable techniques from nonparameteric statistics for real-time robot learning. *Applied Intelligence*, 17:49–60, 2002.

[11] Christopher M Bishop. *Pattern recognition and machine learning*, volume 4. Springer, New York, NY, 2006.

[12] Manuel Muhlig, Michael Gienger, Sven Hellbach, Jochen J. Steil, and Christian Goerick. Task-level imitation learning using variance-based movement optimization. *2009 IEEE International Conference on Robotics and Automation*, pages 1177–1184, 2009.

[13] Sylvain Calinon, Florent D'halluin, Eric L Sauser, Darwin G Caldwell, and Aude G Billard. Learning and reproduction of gestures by imitation. *Robotics & Automation Magazine, IEEEDarwin*, 17(2):44–54, 2010.

[14] Peter Pastor, Mrinal Kalakrishnan, Franziska Meier, Freek Stulp, Jonas Buchli, Evangelos Theodorou, and Stefan Schaal. From dynamic movement primitives to associative skill memories. *Robotics and Autonomous Systems*, 61(4):351–361, 2013.

[15] Merlin Donald. *Origins of the modern mind: Three stages in the evolution of culture and cognition*. Harvard University Press, 1991.

[16] Giacomo Rizzolatti, Leonardo Fogassi, and Vittorio Gallese. Neurophysiological mechanisms underlying the understanding and imitation of action. *Nature Reviews Neuroscience*, 2(9):661–670, 2001.

[17] Yuri a. Ivanov and Aaron F. Bobick. Recognition of visual activities and interactions by stochastic parsing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):852–872, 2000.

[18] Volker Krüger, Danica Kragic, Aleš Ude, and Christopher Geib. The meaning of action: a review on action recognition and mapping. *Advanced Robotics*, 21(13):1473–1501, 2007.

[19] Geir E Hovland, Pavan Sikka, and Brenan J McCarragher. Skill Acquisition from Human Demonstration Using a Hidden Markov Model. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, pages 2706–2711, 1996.

[20] Monica N Nicolescu and Maja J Mataric. A Hierarchical Architecture for Behavior-Based Robots. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 227—-233, 2002.

[21] Michael Pardowitz, Steffen Knoop, Ruediger Dillmann, and Raoul D. Zollner. Incremental learning of tasks from user demonstrations, past experiences, and vocal comments. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 37(2):322–332, 2007.

[22] S. Ekvall and D. Kragic. Learning Task Models from Multiple Human Demonstrations. *ROMAN 2006 - The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pages 358–363, 2006.

[23] Simon Manschitz, Jens Kober, Michael Gienger, and Jan Peters. Learning to Sequence Movement Primitives from Demonstrations. In *International Conference on Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJI*, pages 4414–4421, 2014.

[24] Isabel Serrano Vicente, Ville Kyrki, Danica Kragic, and Martin Larsson. Action recognition and understanding through motor primitives. *Advanced Robotics*, 21:1687–1707, 2007.

[25] Volker Kruger, Dennis Herzog, Sanmohan Baby, Ales Ude, and Danica Kragic. Learning actions from observations. *IEEE Robotics & Automation Magazine*, 17(2):30–43, 2010.

[26] Monica N Nicolescu and Maja J Mataric. Natural Methods for Robot Task Learning : Instructive Demonstrations , Generalization and Practice. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 241–248, 2003.

[27] Kyuhwa Lee and Yiannis Demiris. Towards incremental learning of task-dependent action sequences using probabilistic parsing. *2011 IEEE International Conference on Development and Learning, ICDL 2011*, 2011.

[28] Kyuhwa Lee, Yanyu Su, Tae Kyun Kim, and Yiannis Demiris. A syntactic approach to robot imitation learning using probabilistic activity grammars. *Robotics and Autonomous Systems*, 61(12):1323–1334, 2013.

[29] Guoting Jane Chang and Dana Kulić. Robot Task Learning from Demonstration Using Petri Nets. In *2013 IEEE RO-MAN: The 22nd IEEE International Symposium on Robot and Human Interactive Communication Gyeongju, Korea*, pages 31–36, 2013.

[30] Hassane Alla and René David. *Discrete, Continuous, and Hybrid Petri Nets*, volume 08. Springer, 1998.

[31] Jiacun Wang. Petri Nets for Dynamic Event-Driven System Modeling. In Paul A Fishwick, editor, *Handbook of Dynamic System Modeling*, chapter 24. CRC Press, 2006.

[32] Tadao Murata. Petri Nets : Properties , Analysis and Applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

[33] Pedro Lima, Hugo Grkio, Vasco Veiga, Anders Karlsson, Instituto De Sistemas, Instituto Superior T, Torre Norte, a V Rovisco Pais, and Lisboa Codex. Petri Nets for Modeling and Coordination of Robotic Tasks 1 Introduction Petri Net Views of a Robotic Task Model. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, pages 190–195, 1998.

[34] H Costelha and P Lima. Modelling, analysis and execution of robotic tasks using petri nets. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1449–1454, 2007.

[35] R. Zollner, T. Asfour, and R. Dillmann. Programming by demonstration: dual-arm manipulation tasks for humanoid robots. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 479–484, 2004.

[36] Va Ziparo, L Iocchi, and D Nardi. Petri net plans: a formal model for representation and execution of multi-robot plans. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 79–86, 2008.

[37] Kurt Jensen. Coloured petri nets: A high level language for system design and analysis. *Advances in Petri Nets 1990*, 483:342–416, 1991.

[38] Wil Van Der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[39] Wil M P Van Der Aalst and Boudewijn F. Van Dongen. Discovering Petri nets from event logs. In Kurt Jensen, Wil M. P. van der Aalst, Gianfranco Balbo, Maciej Koutny, and Karsten Wolf, editors, *Transactions on Petri Nets and Other Models of Concurrency VII*, pages 372–422. Springer, 2013.

[40] B. F. Van Dongen, A. K. Alves De Medeiros, and L. Wen. Process mining: Overview and outlook of Petri net discovery algorithms. In Kurt Jensen and Wil M. P. van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, pages 225–242. Springer, 2009.

[41] Barrett Technology Inc. *BH8-282 Datasheet*, 2013.

[42] Günter Schreiber, Andreas Stemmer, and Rainer Bischoff. The fast research interface for the kuka lightweight robot. In *IEEE Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications How to Modify and Enhance Commercial Controllers (ICRA 2010)*, 2010.

[43] Philippe Gerum. Xenomai-implementing a rtos emulation framework on gnu/linux. *White Paper, Xenomai*, 2004.

[44] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[45] Steve Cousins. Exponential growth of ROS [ROS topics]. *IEEE Robotics & Automation Magazine*, 1(18):19–20, 2011.

[46] Herman Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.

[47] Libbarrett 1.2.2 documentation. `http://web.barrett.com/libbarrett`. Accessed: 11-07-2015.

[48] Robert A. Moog. Midi: Musical instrument digital interface. *Journal of the Audio Engineering Society*, 34(5):394–404, 1986.

[49] Steinberg Media Technologies GmbH. *Audio Streaming Input Output (ASIO) 2.2 Specification*, 2006.

[50] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.

[51] Ronen Sosnik, Bjoern Hauptmann, Avi Karni, and Tamar Flash. When practice leads to co-articulation: The evolution of geometrically defined movement primitives. *Experimental Brain Research*, 156:422–438, 2004.

[52] Peter Huber, Kurt Jensen, and Robert M Shapiro. Hierarchies in coloured petri nets. In *Advances in Petri Nets 1990*, pages 313–341. Springer, 1991.